UNIVERSITÀ DEGLI STUDI DI VERONA

DIPARTIMENTO DI MANAGEMENT

Corso di Laurea magistrale in

Banca e Finanza

Implementazione del sensitivities-based method (FRTB) con AAD: applicazione ad un portafoglio equity

Relatore
Prof.
Alessandro Gnoatto

Laureando
Michele Ercole
Matricola VR486795

Anno Accademico 2023/2024

Implementazione del sensitivities-based method (FRTB) con AAD: applicazione ad un portafoglio equity

Michele Ercole

Master Thesis

Verona, 2025

Implementazione del sensitivities-based method (FRTB) con AAD: applicazione ad un portafoglio equity

Dipartimento di Scienze Economiche
Universitá di Verona,

presented by

Michele Ercole

Verona, 2025

student ID: VR486795

Verona, March 19th, 2025

Relatore: Prof. Dr. Alessandro Gnoatto
Correlatore: Prof. Dr. Cosimo-Andrea Munari

Alla mia famiglia.

Indice

1 Introduzione **1**

2 FRTB **3**
2.1 Introduzione alla normativa FRTB 3
2.2 I pilastri della normativa FRTB 6
2.2.1 Standardised Approach (SA) 6
2.2.2 Internal Models Approach (IMA) 9
2.3 Focus Sensitivities-Based Method (SBM) 11
2.3.1 Metodologie di calcolo del Delta e Vega Risk 11
2.3.2 Focus su Equity 17

3 Automatic Differentiation **25**
3.1 Introduzione all'Automatic Differentiation (AD) 25
3.2 L'Automatic Differentiation 26
3.2.1 Forward Automatic Differentiation 28
3.2.2 Backward Automatic Differentiation 30
3.2.3 Confronto tra Forward e Backward Differentiation 32
3.2.4 Stochastic Automatic Differentiation 34
3.3 Implementazione all'interno della libreria finmath in Java 38
3.3.1 Concetti chiave 39
3.3.2 Classi principali 40

4 Implementazione in Java **45**
4.1 Introduzione al progetto Java 45
4.2 Analisi del codice 46
4.3 Un esempio di utilizzo 58

5 Conclusioni **65**

Bibliografia **67**

Appendice **69**

1 Introduzione

Questo elaborato nasce dall'esperienza di tirocinio curricolare svolta presso Iason, una società di consulenza specializzata in risk management per i principali istituti bancari italiani ed europei. Durante questo periodo, ho avuto modo di approfondire la normativa FRTB, con particolare attenzione al sensitivities-based method, utilizzato come componente dell'approccio standardizzato per il calcolo dei requisiti di capitale legati al rischio di mercato. Attualmente, questa metodologia si basa sull'impiego delle differenze finite per il calcolo delle sensitivities dei portafogli. Sebbene tale approccio sia apprezzato per la sua immediatezza applicativa e per la standardizzazione della reportistica, esso rappresenta un'approssimazione delle greche calcolate. L'idea alla base di questo studio è stata quindi quella di applicare la metodologia Automatic Differentiation (AD) e, nello specifico, Adjoint Automatic Differentiation (AAD) per il calcolo dei requisiti di capitale, fornendo una soluzione più rigorosa rispetto ai metodi tradizionali. Inoltre, il progetto si propone di riprodurre, oltre al calcolo delle derivate previste, tutte le fasi successive necessarie alla determinazione del requisito finale di capitale, partendo da un insieme ipotetico di trades. Si precisa che, in questo progetto, è stata curata l'implementazione delle disposizioni inerenti la sola classe di rischio equity, per tre tipologie di strumenti. Nello specifico si tratta di opzioni europee, opzioni bermuda ed azioni.
Per lo sviluppo dell'applicazione è stato scelto Java, un linguaggio di programmazione ampiamente utilizzato durante questo corso di laurea magistrale. Grazie alla sua struttura orientata agli oggetti, Java si presta particolarmente bene all'implementazione di questo progetto. Un ruolo fondamentale è stato svolto dalla libreria finmath, che non solo offre un'implementazione dell'AAD, ma include anche classi utili per il pricing di strumenti finanziari.
La struttura dell'elaborato è organizzata nel seguente modo: il primo capitolo introduce la normativa, fornendo le basi teoriche necessarie per i calcoli successivi. Dopo una panoramica generale sulla FRTB, si entra nel dettaglio del Sensitivities-Based Method, approfondendo le metodologie di calcolo dei tre principali pilastri: Delta Risk, Vega Risk e Curvature Risk. Per ognuno di essi vengono presentati i dettagli necessari per il calcolo nell'ambito del rischio equity. Il secondo capitolo si concentra sull'Automatic Differentiation, spiegandone le due varianti principali, Forward e Backward, con un approfondimento su quest'ultima, più adatta all'implementazione pratica dell'elaborato. Infine, nell'ultimo capitolo viene illustrato il codice sviluppato (riportato in appendice) e viene fornito un esempio pratico basato su un portafoglio di strumenti fittizi, dimostrando l'efficacia dell'approccio proposto.

2 FRTB

2.1 Introduzione alla normativa FRTB

Lo scopo di questo capitolo è fornire un quadro informativo della normativa FRTB, introducendo il contesto di applicazione, le innovazioni introdotte e le principali sfide implementative. Vista la destinazione pratica della normativa e il suo contesto di applicazione, caratterizzato da un approccio oggettivo e meno soggetto a possibili interpretazioni, non è scopo di questo capitolo fornirne, ma piuttosto descrivere in maniera fedele quanto stabilito nei documenti ufficiali (3), (4). Un maggiore livello di dettaglio è dedicato alla sezione relativa al *Sensitivities-Based Method* (SBA), con particolare attenzione all'asset class Equity. Quest'ultima è infatti protagonista dell'implementazione sviluppata nel capitolo 3 di questo elaborato, in cui verranno adottate le metodologie riportate di seguito; l'unica eccezione riguarderà il calcolo delle *sensitivities*. In particolare, al posto del metodo delle differenze finite, verrà utilizzata l'*Adjoint Automatic Differentiation* (AAD), così come descritta dal capitolo 2 di questa tesi.

Origine e motivazioni

La normativa FRTB vede la prima pubblicazione nel 2016, rappresentando una pietra miliare nel panorama della regolamentazione bancaria internazionale. Precedentemente, Basilea 2.5, introdotta in risposta alla crisi finanziaria del 2007-2008, aveva rafforzato i requisiti di capitale per il rischio di mercato (i.e. Il rischio di perdite derivanti da movimenti nei prezzi di mercato (3)), ma non era riuscita a risolvere appieno le criticità strutturali del sistema regolamentare, lasciando margini per pratiche di arbitraggio regolamentare. In questo contesto, FRTB si colloca come una risposta più completa e sistematica. Tuttavia, la sfida implementativa di questo approccio ha portato alla necessità di una revisione nel 2019 e all'introduzione di una finestra temporale di applicazione per gli istituti bancari. Questo ha permesso un'adozione graduale e strutturata delle nuove regole, considerando l'ampio ambito di applicazione e impatti significativi sul calcolo dei requisiti patrimoniali. FRTB riguarda principalmente le banche e le istituzioni finanziarie con esposizioni significative ai rischi di mercato. Si applica al trading book, includendo gli strumenti detenuti a fini di negoziazione o per coperture attive, e lo distingue dal banking book, al fine di prevenire arbitraggi regolamentari. Tale distinzione non risultava infatti abbastanza marcata nel precedente quadro regolamentare. Le banche con dimensioni più ridotte e attività di trading limitate possono essere esentate dai requisiti più complessi dell'FRTB e adottare

approcci semplificati (2). L'introduzione di FRTB è stata guidata dalla necessità di risolvere criticità significative identificate nei modelli di gestione del rischio di mercato adottati dalle istituzioni finanziarie. Durante la crisi finanziaria globale, molti modelli interni delle banche si sono rivelati incapaci di catturare adeguatamente i rischi reali, portando a perdite ingenti e a una riduzione della fiducia nel sistema finanziario. Tra le altre motivazioni principali che hanno spinto alla revisione si trova la necessità di migliorare la sensibilità al rischio delle metodologie di calcolo del capitale, di introdurre un approccio standardizzato più granulare e rappresentativo dei rischi reali e, infine, la riduzione delle disparità tra le banche che utilizzano modelli interni e quelle che si affidano a metodi standardizzati, incrementando la trasparenza e la comparabilità.

Obiettivi principali e impatto sulla regolamentazione bancaria

Come anticipato precedentemente, FRTB si pone l'obiettivo primario di rafforzare la resilienza del sistema bancario internazionale nei confronti dei rischi di mercato. Tra gli obiettivi chiave si richiamano: l'introduzione di una netta separazione tra il trading book e il banking book, al fine di evitare pratiche di arbitraggio regolamentare. L'adozione di nuove metriche per il calcolo del capitale, tra cui l'Expected Shortfall (ES), che sostituisce il Value at Risk (VaR) come misura principale del rischio. La promozione di una maggiore coerenza nell'applicazione dei modelli interni attraverso requisiti più stringenti per la loro approvazione. L'introduzione di FRTB, seppur mirata a rafforzare la resilienza del sistema bancario internazionale, ha posto notevoli sfide per le istituzioni finanziarie, in particolare nell'implementazione pratica delle nuove regole. Come descritto in (6), secondo European Banking Authority (EBA), l'applicazione del framework ha richiesto ingenti investimenti tecnologici e operativi, inclusi l'adeguamento dei sistemi di gestione dati e lo sviluppo di capacità interne per la modellazione dei rischi e il calcolo del capitale regolamentare. Un altro aspetto evidenziato da EBA riguarda la necessità di bilanciare l'armonizzazione normativa a livello europeo con l'adattamento delle regole ai contesti locali. Questa sfida riflette l'esigenza di garantire uniformità nell'applicazione del framework, preservando al contempo la comparabilità e la trasparenza tra le istituzioni finanziarie. Inoltre, la complessità delle metodologie introdotte, sia standardizzate sia basate su modelli interni, ha generato incertezza operativa. EBA ha sottolineato come il coinvolgimento attivo di banche, autorità di vigilanza e stakeholder sia cruciale per superare queste difficoltà e promuovere una transizione graduale ed efficace al nuovo regime. Infine, il framework presenta opportunità significative per migliorare la gestione del rischio di mercato e aumentare la trasparenza nel sistema finanziario europeo, ma i costi di implementazione rimangono una delle principali preoccupazioni, con possibili impatti sul bilancio delle istituzioni finanziarie minori.

Trading Book e Banking Book

Come anticipato, una delle principali innovazioni di FRTB è la ridefinizione del confine tra Banking Book e Trading Book, una misura utile per eliminare arbitraggi regolamentari; infatti, nel contesto di Basilea 2.5 le banche avevano una certa flessibilità nel classificare le proprie posizioni all'interno del Banking Book o del Trading Book. Questa discrezionalità consentiva di ottimizzare artificialmente i requisiti patrimoniali, spostando strumenti finanziari tra i due libri in base ai trattamenti regolamentari più favorevoli. Tale distinzione si basa ora su criteri più oggettivi, che riflettono l'intenzione gestionale e le caratteristiche contrattuali degli strumenti finanziari. Di seguito si riporta un breve riassunto delle caratteristiche degli strumenti inclusi nei due gruppi:

- **Strumenti nel Trading Book:**
  - Includono strumenti finanziari detenuti principalmente con l'obiettivo di negoziazione attiva o per protezione (hedging) di altre posizioni.
  - Comprendono titoli azionari, obbligazioni, derivati e strumenti complessi, generalmente valutati a fair value.
  - Le posizioni devono essere gestite con un orizzonte di breve termine e sono soggette a un monitoraggio frequente, generalmente giornaliero.
- **Strumenti nel Banking Book:**
  - Comprendono strumenti finanziari detenuti con l'intento di conservarli fino alla scadenza o per generare flussi di cassa stabili, come prestiti, obbligazioni a lungo termine e investimenti strategici.
  - Non sono destinati alla negoziazione attiva e sono valutati principalmente al costo ammortizzato, salvo eccezioni specifiche.
  - Le variazioni di valore di mercato non incidono direttamente sul conto economico, salvo nei casi in cui siano classificati come strumenti disponibili per la vendita (available-for-sale).

Questa nuova definizione, potenziata maggiormente nella versione revisionata del 2019, richiede che le banche adottino processi interni per documentare la destinazione iniziale degli strumenti e ottengano approvazioni regolamentari per eventuali riclassificazioni, riducendo così la discrezionalità. Tale separazione comporta però sfide operative per le banche, inclusa l'implementazione di sistemi IT avanzati per monitorare e gestire i portafogli separatamente. Inoltre, richiede che i regolatori nazionali forniscano linee guida precise per garantire un'applicazione uniforme delle nuove regole.

2.2 I pilastri della normativa FRTB

2.2.1 Standardised Approach (SA)

Terminologia Chiave

1. *Classi di rischio:* Sono individuate sette classi di rischio principali, ognuna con caratteristiche specifiche:
    - *General Interest Rate Risk* (GIRR): variazioni delle curve dei tassi di interesse.
    - *Credit Spread Risk* (CSR), suddiviso in:
        - *Non-securitisations*.
        - *Securitisations non-CTP*.
        - *Securitisations CTP*.
    - *Equity Risk*: variazioni nei prezzi azionari.
    - *Commodity Risk*: variazioni nei prezzi delle materie prime.
    - *Foreign Exchange Risk* (FX): variazioni nei tassi di cambio.

2. *Fattore di rischio:* Ogni variabile che influenza il valore di uno strumento finanziario, come il prezzo di un'azione o di una materia prima.

3. *Bucket:* Gruppo di fattori di rischio con caratteristiche simili, utilizzato per aggregare le sensibilità all'interno di una classe di rischio.

4. *Posizione di rischio:* Indica la componente del rischio di uno strumento legata a un fattore specifico, con un calcolo che varia in base alla tipologia considerata:
    - Per il *Delta e Vega Risk*, coincide con la sensibilità dello strumento rispetto al fattore di rischio..
    - Per il *Curvature Risk*, essa si basa sulle perdite incrementali calcolate applicando due scenari di stress (positivo e negativo).

5. *Requisito di capitale:* È la quantità di capitale che una banca deve detenere per far fronte ai rischi assunti. Questo requisito viene calcolato come:
    a.) Aggregazione delle *posizioni di rischio* a livello di ciascun bucket (*intra-bucket*).
    b.) Aggregazione tra i *bucket appartenenti alla stessa classe di rischio* (*inter-bucket*).

Definizione

Con *Standardised Approach* (SA) si indica uno dei pilastri fondamentali di FRTB, introdotto dal Comitato di Basilea per la Vigilanza Bancaria (BCBS) per calcolare i requisiti patrimoniali minimi associati al rischio di mercato. L'obiettivo principale del SA è garantire una metodologia chiara, uniforme e granulare, offrendo una misura più rigorosa rispetto al precedente framework. Questa nuova struttura risolve molte delle criticità dei modelli precedenti, limitando la discrezionalità delle banche e migliorando la comparabilità tra istituzioni finanziarie a livello globale.
Tale metodo standardizzato si articola in tre componenti principali:

1. **Sensitivities-Based Method (SBM)**: misura le sensibilità del portafoglio ai principali fattori di rischio come variazioni nei tassi di interesse, nei prezzi delle azioni o negli spread di credito.

2. **Default Risk Capital (DRC)**: cattura il rischio di default improvviso (*jump-to-default*) associato alle esposizioni soggette al rischio di credito.

3. **Residual Risk Add-On (RRAO)**: incorpora i rischi residui legati a strumenti esotici o non modellabili adeguatamente.

Questa struttura consente di ottenere una stima del rischio di mercato più precisa, mitigando il rischio di sottostima delle perdite e garantendo un trattamento consistente degli strumenti finanziari.
La formula generale è data da:

$$CapitalRequirement = SBM + DRC + RRAO \tag{2.1}$$

Il calcolo complessivo prevede l'aggregazione semplice delle tre componenti, ciascuna delle quali viene calcolata separatamente in base a metodologie rigorosamente definite. Di seguito si riportano brevemente le principali caratteristiche di queste tre componenti.

Sensitivities-Based Method (SBM)

Il *Sensitivities-Based Method* (SBM) è la componente centrale di Standardised Approach, per il calcolo dei requisiti patrimoniali minimi relativi al rischio di mercato. Questo metodo misura l'esposizione ai fattori di rischio principali attraverso una valutazione dettagliata delle sensibilità del portafoglio di trading.
SBM è strutturato per catturare l'impatto delle variazioni nei fattori di rischio su tre dimensioni principali:

1. *Delta Risk*: rappresenta la sensibilità lineare del valore dello strumento a piccole variazioni nei fattori di rischio.

2. *Vega Risk*: misura la sensibilità del valore dello strumento alle variazioni nella volatilità implicita dei fattori di rischio.

3. *Curvature Risk*: cattura gli effetti non lineari, particolarmente rilevanti per strumenti derivati con opzionalità, attraverso scenari di shock positivi e negativi.

Il processo di calcolo prevede tre fasi fondamentali:

i) Determinazione delle sensibilità rispetto ai fattori di rischio regolamentari (i.e. *Net Sensitivities*)

ii) Ponderazione delle sensibilità utilizzando coefficienti calibrati sulla base della volatilità storica dei fattori di rischio. (i.e. *Weighted Sensitivities*)

iii) Aggregazione dei risultati *intra-bucket* e *inter-bucket*, considerando correlazioni predefinite. Per riflettere l'incertezza delle correlazioni in condizioni di stress, il calcolo viene ripetuto applicando tre scenari distinti: base, elevata correlazione e bassa correlazione.

Questa metodologia offre un quadro standardizzato e granulare della misurazione del rischio, garantendo risultati più accurati e un trattamento coerente degli strumenti finanziari.

Default Risk Capital (DRC)

Il *Default Risk Capital* (DRC) rappresenta la componente dedicata al rischio di *jump to default* degli strumenti finanziari. Questo rischio è particolarmente rilevante per esposizioni a obbligazioni corporate, sovrane e prodotti strutturati.
Il calcolo del DRC prevede un approccio strutturato come di seguito:

i. Le esposizioni di strumenti soggetti al rischio di default sono classificate in base alla presenza di securitizzazione.

ii. Viene calcolata l'esposizione netta per ciascun bucket, sottraendo le posizioni short dalle posizioni long relative allo stesso emittente.

iii. Vengono applicati pesi specifici per ciascun bucket, calibrati sulla base della probabilità di default.

iv. I risultati vengono aggregati per ottenere il requisito complessivo.

Residual Risk Add-On (RRAO)

Il *Residual Risk Add-On* (RRAO) è stato introdotto per catturare i rischi residui derivanti da strumenti con caratteristiche esotiche o non standardizzate.
Gli strumenti soggetti al RRAO includono:

- Opzioni esotiche
- Derivati complessi
- Altri strumenti non lineari, con caratteristiche definite dalla normativa

Il requisito di capitale è calcolato come una somma semplice dei *gross notional* moltiplicati per un peso fisso, definito dalla normativa.

2.2.2 Internal Models Approach (IMA)

L'*Internal Models Approach* (IMA) rappresenta un metodo avanzato per il calcolo dei requisiti patrimoniali minimi associati al rischio di mercato. Tale approccio consente alle banche di utilizzare modelli interni per stimare l'esposizione al rischio, offrendo maggiore flessibilità rispetto al metodo standardizzato. Tuttavia, l'utilizzo di tali modelli richiede un processo rigoroso di validazione e di approvazione da parte delle autorità di vigilanza, con controlli costanti per garantire l'affidabilità delle misurazioni del rischio.
La regolamentazione stabilisce criteri stringenti per l'approvazione, il monitoraggio e il mantenimento dell'IMA, con l'obiettivo di ridurre i rischi di errore e migliorare la stabilità complessiva del sistema finanziario. La logica di tale controllo è data dalla perdita del requisito di standardizzazione dovuta alla mancanza di un approccio definito dal regolatore.
Per adottare l'IMA, le banche devono soddisfare una serie di requisiti qualitativi e operativi, che riguardano sia l'affidabilità dei modelli sia la capacità organizzativa dell'istituzione finanziaria. L'autorità di vigilanza svolge un ruolo chiave nella valutazione iniziale e nel monitoraggio continuo dell'implementazione.
In primo luogo, la banca deve dimostrare di disporre di un sistema di gestione del rischio robusto dal punto di vista concettuale e operativo. Questo implica che il modello utilizzato sia stato validato accuratamente e che sia in grado di misurare con precisione le perdite attese e inattese derivanti dal rischio di mercato. Tale processo di validazione include l'analisi approfondita delle ipotesi sottostanti al modello e dei risultati generati in condizioni di mercato ordinarie e di stress.
Un ulteriore requisito riguarda l'organizzazione interna. La gestione dei modelli deve essere affidata a unità indipendenti di controllo del rischio, che devono produrre report giornalieri sull'esposizione complessiva della banca ai fattori di rischio e monitorare costantemente i limiti stabiliti. Viene anche stabilito dalla normativa, che la presenza di un team

con competenze adeguate in materia di modellizzazione, gestione del rischio e validazione è indispensabile per mantenere l'affidabilità del modello nel tempo.

Un elemento fondamentale per l'approvazione dell'IMA è la capacità della banca di eseguire *backtesting* e *Profit & Loss Attribution* (PLA). Il primo consiste nel confronto tra le perdite effettive osservate e quelle previste dal modello interno. Tale processo permette di verificare l'accuratezza storica delle stime del rischio e di identificare eventuali discrepanze. Il PLA, invece, misura la capacità del modello di spiegare correttamente le variazioni nei profitti e nelle perdite del *trading desk*. Se un desk fallisce ripetutamente il test PLA compromette, per la banca, la possibilità di utilizzare l'IMA e ne consegue l'obbligatorio utilizzo dell'approccio SA.

Un altro requisito riguarda la capacità della banca di eseguire *stress test* regolari. Questi test sono progettati per valutare la vulnerabilità del portafoglio a scenari di mercato estremi e sono particolarmente importanti per comprendere i rischi associati a eventi rari ma di grande impatto. I risultati devono essere rivisti periodicamente dal senior management e utilizzati per prendere decisioni strategiche in materia di capitale e gestione del rischio.

Uno degli elementi chiave di IMA è l'utilizzo dell'*Expected Shortfall* (ES) come misura principale del rischio di mercato, in sostituzione del tradizionale *Value-at-Risk* (VaR). L'ES rappresenta la perdita media nei casi in cui questa superi il VaR, offrendo una misura più conservativa e accurata del rischio nelle code della distribuzione. L'introduzione dell'*Expected Shortfall* ha migliorato la capacità delle banche di misurare il rischio di mercato, rendendo i requisiti patrimoniali più sensibili alle condizioni di stress.

Per essere inclusi nei modelli interni, i fattori di rischio devono superare il *Risk Factor Eligibility Test* (RFET). Questo test verifica che i fattori siano modellabili, ossia osservati con sufficiente frequenza nei mercati. In particolare, un fattore di rischio è considerato modellabile se è osservato almeno 24 volte in un anno, con una frequenza minima mensile. Se un fattore di rischio non soddisfa questo requisito, viene classificato come non modellabile (*Non-Modellable Risk Factor*, NMRF) e viene trattato separatamente attraverso l'applicazione di *add-on* specifici. Questo meccanismo garantisce che i modelli interni siano basati su dati robusti e riflettano accuratamente le condizioni di mercato.

L'*Internal Models Approach* rappresenta uno strumento avanzato per il calcolo dei requisiti patrimoniali legati al rischio di mercato, offrendo alle banche la possibilità di sviluppare modelli personalizzati e più aderenti alle proprie esposizioni contenute nei libri contabili. Tuttavia, l'implementazione dell'IMA richiede il rispetto di requisiti rigorosi, sia in termini di validazione dei modelli sia in termini di governance interna. L'introduzione di strumenti come l'Expected Shortfall e il Risk Factor Eligibility Test ha reso l'IMA più solido e affidabile, migliorando la capacità delle banche di misurare e mitigare il rischio in contesti complessi e dinamici.

2.3 Focus Sensitivities-Based Method (SBM)

Come accennato in precedenza, il Sensitivities-Based Method (SBM) utilizza le sensibilità degli strumenti finanziari a un insieme predefinito di fattori di rischio per calcolare i requisiti di capitale associati ai rischi Delta, Vega e Curvature. Queste sensibilità vengono ponderate con specifici pesi di rischio e successivamente aggregate in due fasi:

a.) All'interno del medesimo bucket, ovvero un gruppo di fattori di rischio con caratteristiche comuni (i.e Aggregazione intra-bucket).

b.) Tra bucket appartenenti alla stessa classe di rischio (i.e Aggregazione inter-bucket).

2.3.1 Metodologie di calcolo del Delta e Vega Risk

È utile ricordare che le banche devono utilizzare modelli di pricing approvati e convalidati per calcolare le sensibilità e i requisiti patrimoniali secondo il Sensitivities-Based Method. Questi modelli devono essere allineati alle pratiche di reporting dei profitti reali e rispettare standard di valutazione prudente per garantire affidabilità e conformità normativa.
La prima fase del calcolo consiste nella misurazione delle sensibilità degli strumenti finanziari ai fattori di rischio, che devono essere espresse nella valuta di reporting della banca.
In questo elaborato si riporta la trattazione riguardante il rischio Equity, indicandone le specificità in 2.3.2.

Calcolo delle Delta e Vega risk

Per il calcolo del *Delta Risk*, la normativa (3) prevede il calcolo della sensibilità tramite il metodo delle differenze finite, con shifts espressamente indicati dalla normativa in base al fattore di rischio.

$$s_k = \frac{V_i(RF_k + h) - V_i(RF_k)}{h} \tag{2.2}$$

Dove:

- s_k: indica la sensibilità rispetto al fattore di rischio k.
- V_i: rappresenta il valore dello strumento finanziario.
- RF_k: è il valore del fattore di rischio considerato.
- h: Lo shift indicato dalla normativa.

Il *Vega Risk* misura la variazione nel valore di mercato di uno strumento finanziario, come un'opzione, in risposta a piccoli cambiamenti nella volatilità implicita (σ_i) del fattore di rischio sottostante. La formula per calcolare la sensibilità s_k è data da:

$$s_k = \frac{\partial V_i}{\partial \sigma_k} \cdot \sigma_k \tag{2.3}$$

Dove:

- $s_{i,k}$: indica la sensibilità rispetto al fattore di rischio k.
- $\frac{\partial V}{\partial \sigma_i}$: rappresenta il vega, ovvero la sensibilità del valore dello strumento i rispetto alla volatilità implicita del fattore di rischio k.
- σ_i: rappresenta la volatilità implicita.

In particolare, la normativa (7) definisce la sensibilità s_k di un'opzione come:

$$s_k = \frac{V_i(1.01 \cdot \sigma_k, x, y) - V_i(\sigma_k, x, y)}{0.01} \tag{2.4}$$

Con:

- σ_k: valore della volatilità implicita k.
- x, y: altri fattori di rischio da cui dipende la funzione di *pricing* utilizzata.

Si tratta dunque di un'approssimazione del vega di un'opzione con il metodo delle differenze finite, applicando uno shock relativo dell'1% al valore originale delle volatilità implicita utilizzata nella funzione di pricing.

$$\frac{\partial V_i}{\partial \sigma_k} \cdot \sigma_k \approx \frac{V_i(1.01 \cdot \sigma_k, x, y) - V_i(\sigma_k, x, y)}{0.01}$$

I valori di vega e della volatilità implicita devono essere derivati da modelli di pricing approvati dall'unità di controllo del rischio indipendente della banca.
Per quanto riguarda le assunzioni, per le classi di rischio tasso (GIRR) e credito (CSR), è possibile utilizzare distribuzioni log-normali o normali. Tuttavia, per le classi Equity, Commodity e FX, è obbligatoria l'adozione della distribuzione log-normale.
Infine, si evidenzia che tutte le sensibilità vega devono essere calcolate escludendo l'impatto degli aggiustamenti di valutazione creditizia (CVA).

Le sensibilità vengono nettate per ciascun risk factor, combinando le esposizioni positive e negative all'interno del portafoglio:

$$s_k = \sum_i \mathrm{s}_{i,k} \tag{2.5}$$

Dove:

- s_k: rappresenta la sensibilità netta al fattore di rischio k.
- $s_{i,k}$: indica la sensibilità di ciascuno strumento *i* nel portafoglio, rispetto al fattore di rischio ***k***.

Successivamente, queste vengono ponderate utilizzando dei pesi di rischio predefiniti (RW_k), calibrati in base alla volatilità storica e alla pericolosità del fattore di rischio. La formula è la seguente:

$$WS_k = RW_k \cdot s_k \tag{2.6}$$

Dove:

- WS_k: rappresenta la sensibilità ponderata per il fattore k.
- RW_k: è il peso di rischio assegnato dalla normativa per il fattore .

a) Aggregazione all'interno dei Bucket: all'interno di ciascun bucket – definito appunto come un gruppo di fattori di rischio con caratteristiche simili – le sensibilità ponderate (WS_k) vengono combinate applicando coefficienti di correlazione regolamentari (ρ_{kl}). La formula per l'aggregazione è:

$$K_b = \sqrt{max(0, \sum_k WS_k^2 + \sum_k \sum_{l \neq k} (\rho_{kl} WS_k WS_l))} \tag{2.7}$$

Dove:

- K_b: requisito di capitale per il bucket b.
- WS_k e WS_l: sensibilità ponderate per i fattori k e l.
- ρ_{kl}: coefficiente di correlazione per i fattori k e l, definito in base al bucket di appartenenza.

b) Aggregazione tra i Bucket: i requisiti di capitale aggregati a livello di singolo bucket vengono poi combinati per ottenere il requisito complessivo per ciascuna classe di rischio. In questa fase, vengono utilizzati coefficienti di correlazione tra bucket (γ_{bc}), secondo la seguente formula:

$$K_{\text{class}} = \sqrt{\sum_b K_b^2 + \sum_b \sum_{c \neq b} \gamma_{bc} S_b S_c} \tag{2.8}$$

Dove:

- K_{class}: requisito di capitale per la classe di rischio.
- γ_{bc}: coefficiente di correlazione tra bucket b e c.
- $S_b = \sum_k WS_k$ per tutti i fattori di rischio nel bucket b
- $S_c = \sum_k WS_k$ per tutti i fattori di rischio nel bucket c

Se i valori S_b e S_c risultano negativi, si applica la seguente alternativa, in cui:

- $S_b = \max\left[\min\left(\sum_k WS_k, K_b\right), -K_b\right]$ per tutti i fattori di rischio nel bucket b.
- $S_c = \max\left[\min\left(\sum_k WS_k, K_c\right), -K_c\right]$ per tutti i fattori di rischio nel bucket c.

Calcolo del Curvature Risk

Il *Curvature Risk* cattura le variazioni non lineari nei valori di mercato degli strumenti finanziari, causate da significativi shock ai fattori di rischio regolamentari. Anche in questo caso la normativa prevede un approccio strutturato per calcolare tale requisito di capitale, che include i seguenti passaggi:

1. Determinazione dello Shock: per ogni fattore di rischio k, si calcolano due scenari di shock, uno positivo ($x_k^{RW^{\text{Curvature}+}}$) e uno negativo ($x_k^{RW^{\text{Curvature}-}}$). Per ciascuno strumento i, il valore post-shock è confrontato con il valore iniziale, considerando il delta ponderato del medesimo strumento:

$$CVR_k^+ = -\sum_i \left\{ V_i\left(x_k^{RW^{\text{Curvature}+}}\right) - V_i(x_k) - RW_k^{\text{Curvature}} \times s_{ik} \right\} \tag{2.9}$$

$$CVR_k^- = -\sum_i \left\{ V_i\left(x_k^{RW^{\text{Curvature}-}}\right) - V_i(x_k) + RW_k^{\text{Curvature}} \times s_{ik} \right\} \tag{2.10}$$

Dove:

- $V_i\left(x_k^{RW^{\text{Curvature}+}}\right)$: Valore dello strumento i dopo l'applicazione dello shock positivo $RW_k^{\text{Curvature}+}$ sul fattore di rischio k.

- $V_i(x_k)$: Valore dello strumento i al livello corrente del fattore di rischio k, senza alcuno shock applicato.
- $RW_k^{\text{Curvature}}$: Peso di rischio per il fattore k associato al rischio di curvatura, come definito dalla normativa.
- s_{ik}: Sensibilità delta dello strumento i rispetto al fattore di rischio k.

2.Aggregazione Intra-Bucket: l'aggregazione del rischio di curvatura all'interno di ciascun bucket avviene utilizzando i coefficienti di correlazione prescritti ρ_{kl}, come specificato nella seguente formula:

$$K_b = \max\left(K_b^+, K_b^-\right), \tag{2.11}$$

dove:

$$K_b^+ = \sqrt{\max\left(0, \sum_k \max(CVR_k^+, 0)^2 + \sum_k \sum_{l \neq k} \rho_{kl} CVR_k^+ CVR_l^+ \psi(CVR_k^+, CVR_l^+)\right)} \tag{2.12}$$

$$K_b^- = \sqrt{\max\left(0, \sum_k \max(CVR_k^-, 0)^2 + \sum_k \sum_{l \neq k} \rho_{kl} CVR_l^- CVR_k^- \psi(CVR_k^-, CVR_l^-)\right)} \tag{2.13}$$

Dove:

- K_b: Requisito di capitale aggregato per il bucket b. Questo viene selezionato come il maggiore tra K_b^+ e K_b^-:
  - (i) $K_b = K_b^+$: Quando lo scenario positivo (upward) è selezionato.
  - (ii) $K_b = K_b^-$: Quando lo scenario negativo (downward) è selezionato.
  - (iii) In caso di uguaglianza ($K_b^+ = K_b^-$), lo scenario positivo viene scelto se $\sum_k CVR_k^+ > \sum_k CVR_k^-$; altrimenti, lo scenario negativo è selezionato.
- K_b^+: Capitale richiesto nello scenario positivo.
- K_b^-: Capitale richiesto nello scenario negativo.

- CVR_k^+ e CVR_k^-: Componenti del rischio di curvatura per il fattore di rischio k, rispettivamente per gli scenari positivo e negativo.
- ρ_{kl}: Coefficiente di correlazione tra i fattori di rischio k e l all'interno dello stesso bucket.
- $\psi(CVR_k, CVR_l)$: Funzione che assume valore 0 se CVR_k e CVR_l hanno entrambi segno negativo, e valore 1 altrimenti.

3.Aggregazione Inter-Bucket: come avviene il delta e per il vega, anche le posizioni di rischio di curvatura devono essere aggregate tra i bucket all'interno di ciascuna classe di rischio, utilizzando le corrispondenti correlazioni γ_{bc}, dove:

$$k_{\text{class}} = \sqrt{\max\left(0, \sum_b K_b^2 + \sum_b \sum_{c \neq b} \gamma_{bc} S_b S_c \psi(S_b, S_c)\right)} \tag{2.14}$$

con:

(a) $S_b = \sum_k CVR_k^+$ per tutti i fattori di rischio nel bucket b, quando lo scenario di shock positivo è stato selezionato per il bucket b. Altrimenti, $S_b = \sum_k CVR_k^-$.

(b) $\psi(S_b, S_c)$ assume il valore 0 se S_b e S_c hanno entrambi segno negativo e 1 altrimenti.

Aggregazione del Requisito di Capitale Totale: per affrontare il rischio che le correlazioni aumentino o diminuiscano nei periodi di stress finanziario, l'aggregazione dei requisiti di capitale a livello intra-bucket e a livello inter-bucket per il rischio delta, vega e curvature, deve essere ripetuta considerando tre diversi scenari che modificano i parametri di correlazione ρ_{kl} e γ_{bc}.

(1) Scenario di correlazione media: i parametri di correlazione ρ_{kl} e γ_{bc}, definiti dalla normativa in maniera specifica per ogni classe di rischio, si applicano direttamente.

(2) Scenario di correlazione alta: i parametri di correlazione ρ_{kl} e γ_{bc} specificati sono uniformemente moltiplicati per 1.25, con un limite massimo del 100%.

$$\rho_{kl}^{high} = \min(1.25 \times \rho_{kl}, 1.0)$$
$$\gamma_{bc}^{high} = \min(1.25 \times \gamma_{bc}, 1.0)$$

(3) Scenario di correlazione bassa: i parametri di correlazione ρ_{kl} e γ_{bc} vengono sostituiti come segue:

$$\rho_{kl}^{low} = \max(2 \times \rho_{kl} - 100\%, 75\% \times \rho_{kl})$$
$$\gamma_{bc}^{low} = \max(2 \times \gamma_{bc} - 100\%, 75\% \times \gamma_{bc})$$

Il requisito finale per la classe di rischio è dato da:

$$K_{finale} = \max\{K_{medium}, K_{high}, K_{low}\} \tag{2.15}$$

Requisito di capitale totale: il requisito di capitale totale viene aggregato come segue:

(1) Per ciascuno dei tre scenari di correlazione, la banca deve sommare i requisiti di capitale separatamente calcolati per delta, vega e curvature, in tutte le classi di rischio, per determinare il requisito di capitale complessivo per quello scenario.

(2) Il requisito di capitale secondo il metodo basato sulle sensibilità è il massimo tra i tre scenari.

   a) Per il calcolo dei requisiti di capitale con l'approccio standardizzato, il requisito di capitale viene determinato per tutti gli strumenti e per tutti i trading desk.

   b) Per il calcolo dei requisiti di capitale per ciascun trading desk, considerando ogni desk come se fosse un portafoglio regolamentare *standalone*, i requisiti di capitale sotto ciascun scenario di correlazione sono calcolati e confrontati a livello di trading desk, e il massimo per ciascun trading desk viene assunto come requisito di capitale.

2.3.2 Focus su Equity

Di seguito vengono riportate le linee guida per l'applicazione del Sensitivities-Based Method (SBM) limitatamente alla classe di rischio equity. Si specifica che anche queste direttive sono contenute in (3).

Delta Equity

I fattori di rischio per il *Delta Equity* includono:

- Prezzi spot delle azioni: misurano la sensibilità del portafoglio ai cambiamenti diretti nei prezzi di mercato delle azioni sottostanti.
- Equity repo rates: catturano il rischio associato ai tassi impliciti utilizzati nelle operazioni di pronti contro termine sulle azioni. [Tale calcolo non è trattato in questo elaborato].

Il delta equity spot rappresenta una misura della sensibilità del valore di mercato di uno strumento finanziario rispetto alle variazioni nei prezzi spot delle azioni (EQ_k). Questo calcolo avviene applicando uno shock dell'1% al prezzo spot dell'azione e valutando l'impatto di questa variazione sul valore dello strumento finanziario (V_i). Successivamente,

la differenza tra il valore dopo la variazione e il valore iniziale viene divisa per 0.01. La formula utilizzata per determinare la sensibilità è la seguente:

$$s_k = \frac{V_i(1.01 \cdot EQ_k) - V_i(EQ_k)}{0.01} \tag{2.16}$$

Dove:

- EQ_k: è il prezzo corrente dell'azione k.
- V_i: rappresenta il valore di mercato dello strumento i, che dipende dal prezzo dell'azione EQ_k.

Per calcolare le sensibilità ponderate (WS_k), i pesi di rischio relativi alle sensibilità ai prezzi spot delle azioni e ai repo rates per i bucket dall'1 al 13 sono specificati nella normativa e riassunti in questa sezione all'interno della tabella 2.2.

Vega Equity

Il *Vega Equity* si concentra sulla sensibilità alle variazioni della volatilità implicita delle opzioni che fanno riferimento ai prezzi spot delle azioni sottostanti. Le caratteristiche principali includono:

- Volatilità implicita (σ): misura la volatilità associata alle opzioni basate sui prezzi azionari.
- Scadenze (*maturity tenors*): la volatilità è calcolata per opzioni con scadenze mappate su una o più delle seguenti durate:
  - 0,5 anni
  - 1 anno
  - 3 anni
  - 5 anni
  - 10 anni
- Esclusione dei tassi repo.
- Approccio obbligatorio: log-normale.

Curvature Equity

Il *Curvature Risk*, per quanto riguarda la classe equity, riguarda i prodotti con opzionalità e cattura le variazioni non lineari associate ad importanti movimenti dei sottostanti. In particolare il calcolo interessa solo i prezzi spot. Al contrario, in questo calcolo vengono esclusi i tassi repo per cui, analogamente al vega, non vi è alcun requisito patrimoniale aggiuntivo da considerare. Di seguito si riportano alcuni dettagli implementativi definiti dalla normativa, per precisare il calcolo presentato nella sezione precedente.

- *Shock*: l'entità dello shock applicato al fattore di rischio considerato, è pari al corrispondente peso di rischio delta; si tratta dei pesi di rischio contenuti in tabella 2.2.
- *Bucket*: i bucket utilizzati per il calcolo del requisito di capitale per il rischio di delta vengono replicati per la valutazione del rischio di curvatura.
- *Correlazione Intra-Bucket*: Per aggregare le posizioni di rischio di curvatura all'interno di un bucket, i parametri di correlazione per il rischio di curvatura ρ_{kl} sono determinati elevando al quadrato i corrispondenti parametri di correlazione per il rischio delta.
- *Correlazione Inter-Bucket*: come nel caso dell'aggregazione all'interno dello stesso bucket, per aggregare le posizioni di rischio di curvatura tra diversi bucket, i parametri di correlazione per il rischio di curvatura γ_{bc} sono determinati elevando al quadrato i corrispondenti parametri di correlazione per il rischio Delta.

Classificazione dei Bucket per Equity

Il rischio equity è suddiviso in bucket che raggruppano strumenti finanziari con caratteristiche comuni. Questi gruppi sono classificati per:

- Capitalizzazione di mercato, definita come la somma delle capitalizzazioni di mercato calcolate in base al valore di mercato del totale delle azioni in circolazione emesse dalla stessa entità legale quotata o da un gruppo di entità legali su tutti i mercati azionari a livello globale. Essa è divisa in:
  - Grande capitalizzazione (i.e maggiore o uguale a USD 2 billion).
  - Piccola capitalizzazione (i.e. minore di USD 2 billion).
- Area geografica, ovvero:
  - Economie sviluppate (i.e. Canada, Stati Uniti, Messico, area euro, paesi dell'Europa occidentale non appartenenti all'area euro (Regno Unito, Norvegia, Svezia, Danimarca e Svizzera), Giappone, Oceania (Australia e Nuova Zelanda), Singapore e Hong Kong SAR).

  - Economie emergenti.

- Settore industriale: ad esempio, tecnologia, energia, beni di consumo.

L'obiettivo di questa classificazione è garantire che il calcolo delle sensibilità sia uniforme e rifletta accuratamente la natura dei rischi. Nella tabella 2.1 è possibile trovare la classificazione indicata da FRTB.

Numero Bucket	Capitalizzazione di mercato	Economia	Settore
1	Grande	Economia emergente	Beni di consumo e servizi, trasporti e magazzinaggio, attività di supporto amministrativo e servizi di supporto, sanità, utility
2			Telecomunicazioni, industria
3			Materiali di base, energia, agricoltura, manifattura, estrazione mineraria e cave
4			Settore finanziario, inclusi finanziamenti garantiti dal governo, attività immobiliari, tecnologia
5	Grande	Economia avanzata	Beni di consumo e servizi, trasporti e magazzinaggio, attività di supporto amministrativo e servizi di supporto, sanità, utility
6			Telecomunicazioni, industria
7			Materiali di base, energia, agricoltura, manifattura, estrazione mineraria e cave
8			Settore finanziario, inclusi finanziamenti garantiti dal governo, attività immobiliari, tecnologia
9	Piccola	Economia emergente	Tutti i settori descritti nei bucket 1, 2, 3 e 4
10		Economia avanzata	Tutti i settori descritti nei bucket 5, 6, 7 e 8
11	Altro settore		
12	Indici azionari di economie avanzate con grande capitalizzazione (non specifici per settore)		
13	Altri indici azionari (non specifici per settore)		

Tabella 2.1: Classificazione dei bucket in base a: capitalizzazione di mercato, economia e settore

Pesi di Rischio (Risk Weights)

Ogni bucket è associato a un peso di rischio (RW_k) che rappresenta la sensibilità del portafoglio agli shock di mercato, così come indicato nella tabella 2.2 per il rischio delta e in tabella 2.3 per il rischio vega. Ad esempio, ai mercati sviluppati vengono associati dei pesi generalmente più bassi, data la stabilità di questi mercati. Al contrario, per i Mercati emergenti si hanno pesi più elevati, riflettendo la maggiore volatilità e il rischio intrinseco di questi mercati. In particolare, i pesi di rischio riguardanti il Vega includono una componente per il rischio di liquidità definita da un orizzonte di liquidità i.e. *Liquidity Horizon* LH_k e vengono calcolati come $RW_k = \min\left(RW_\sigma \cdot \frac{\sqrt{LH_k}}{\sqrt{10}}; 100\%\right)$ con RW_σ pari a 55%.

Bucket	RW delta
1	55%
2	60%
3	45%
4	55%
5	30%
6	35%
7	40%
8	50%
9	70%
10	50%
11	70%
12	15%
13	25%

Tabella 2.2: Pesi di rischio delta per i bucket da 1 a 13

Bucket	Liquidity Horizon	RW Vega
1,2,3,4,5,6,7,8,12,13	20%	77.78%
9,10,11	60%	100%

Tabella 2.3: Pesi di rischio vega per i bucket da 1 a 13

a) Aggregazione all'interno dei Bucket: all'interno di ciascun bucket, i fattori di rischio sono correlati attraverso parametri predefiniti (ρ_{kl}). La correlazione tra sensibilità ponderate (WS_k e WS_l) segue regole specifiche, in particolare per il rischio delta si utilizzano i seguenti coefficienti:

(1) Il parametro di correlazione ρ_{kl} è fissato al 99.90% se:

   (a) Una sensibilità è al prezzo spot dell'equity e l'altra ai tassi repo dell'equity;

   (b) Entrambe sono riferite allo stesso nome di emittente equity.

(2) Il parametro di correlazione ρ_{kl} è determinato come segue nei casi in cui entrambe le sensibilità sono al prezzo spot dell'equity:

   (a) 15% tra due sensibilità all'interno dello stesso bucket appartenenti ai bucket numero 1, 2, 3 o 4;

   (b) 25% tra due sensibilità all'interno dello stesso bucket appartenenti ai bucket numero 5, 6, 7 o 8;

   (c) 7.5% tra due sensibilità all'interno dello stesso bucket appartenenti al bucket numero 9;

(d) 12.5% tra due sensibilità all'interno dello stesso bucket appartenenti al bucket numero 10;

(e) 80% tra due sensibilità all'interno dello stesso bucket appartenenti ai bucket numero 12 o 13.

(3) Lo stesso parametro di correlazione ρ_{kl} definito nei punti (2)(a)-(d) sopra si applica nei casi in cui entrambe le sensibilità sono ai tassi repo.

(4) Le correlazioni sopra indicate non si applicano al bucket del settore "other" (bucket 11). Per questo, avviene quanto segue:

- L'aggregazione delle posizioni di rischio delta e vega segue la semplice somma dei valori assoluti delle sensibilità nette ponderate assegnate a questo bucket.

$$K_b = \sum_k |WS_k|$$

- L'aggregazione delle posizioni di rischio di curvatura utilizza la seguente formula:

$$K_b = \max\left(\sum_k \max(CVR_k^+, 0), \sum_k \max(CVR_k^-, 0)\right)$$

Si specifica inoltre che i coefficienti di correlazioni utili all'aggregazione intra-bucket del vega equity, tranne per il bucket 11, seguono la seguente formula:

$$\rho_{kl} = \min\left(\rho_{kl}^{DELTA} \cdot \rho_{kl}^{optionmaturity}; 1\right)$$

con:

$$\rho_{kl}^{optionmaturity} = e^{-\alpha \cdot \frac{|T_k - T_l|}{min(T_k; T_l)}} \tag{2.17}$$

dove:

- α= 1%
- T_k (T_l) è la *maturity* dell'opzione k (l) di cui si sta calcolando il vega

Infine, per l'aggregazione intra-bucket del rischio di curvatura, si utilizza la correlazione del rischio delta elevata al quadrato.

b) Aggregazione tra i Bucket: quando si combinano i bucket da 1 a 13, il parametro di correlazione (γ_{bc}) per il rischio delta e vega varia come segue:

- 15% se entrambi i bucket appartengono ai bucket da 1 a 10.

- 0% se uno dei bucket è il bucket 11.
- 75% se uno dei bucket è il 12 e l'altro il 13.
- 45% in tutti gli altri casi.

Per il rischio di curvatura vale quanto detto sopra, ovvero i coefficienti utilizzati per il rischio delta e vega, appena riportati, vengono considerati elevandoli al quadrato.

3 Automatic Differentiation

3.1 Introduzione all'Automatic Differentiation (AD)

L'*Automatic Differentiation* (AD) è una tecnica algoritmica che consente di calcolare in modo efficiente e preciso le derivate di una funzione implementata in un calcolatore. Ciò che contraddistingue questa metodologia è la capacità di fornire derivate esatte, senza ricorrere a metodi approssimativi come il *bump and recalc* (i.e. differenze finite), che si basa sull'applicazione di shock infinitesimali ai parametri. L'AD si fonda sul principio che qualsiasi funzione può essere decomposta in una sequenza di operazioni elementari (ad esempio, somme, prodotti, funzioni esponenziali o logaritmi), per le quali le regole di derivazione sono ben definite. Seguendo il flusso di calcolo del programma, AD propaga le derivate attraverso queste operazioni in modo sistematico e strutturato, ottenendo così le derivate rispetto agli input in maniera esatta ed efficiente. Una modalità frequente per descrivere il processo di calcolo delle derivate tramite AD è la rappresentazione sotto forma di *Directed Acyclic Graph* (DAG). In tale grafo, i nodi corrispondono alle operazioni elementari e gli archi rappresentano i flussi di dati tra un'operazione e la successiva. L'AD attraversa quindi questo DAG secondo una precisa sequenza, calcolando e propagando le derivate in ciascun nodo fino a ottenere le derivate rispetto a tutte le variabili di input.

Le origini della Differenziazione Automatica risalgono al lavoro di Wengert (1964) (15), che ha posto le basi per l'applicazione dell'AD in numerosi campi. Nel corso degli anni, questa tecnica è stata affinata e adottata in diversi ambiti scientifici e ingegneristici, come dimostrato da Giering e Kaminski (1998) (11), che hanno evidenziato il ruolo cruciale dell'AD nell'ottimizzazione e nell'analisi di sensibilità. Successivamente, in campo finanziario, lavori come quello di Capriotti e Giles (2011)(5) hanno ulteriormente sviluppato e ottimizzato l'applicazione dell'AD, rendendo possibile il calcolo delle cosiddette *Adjoint Greeks* in maniera efficiente. Studi più recenti, ad esempio Antonov (2017)(1), hanno esteso l'ambito di applicazione al calcolo delle sensibilità a strumenti *callable*. Allo stesso modo, Henrard (2014) (12) ha approfondito l'utilizzo della Differenziazione Automatica applicata ad alcuni modelli finanziari, fornendo un'ottima base di studio, anche per questo elaborato. Quest'ultimo si basa principalmente sui lavori di Christian Fries riguardanti l'AD, ovvero (9, 8, 10), riprendendone la notazione ed i concetti chiave. Analogamente, l'implementazione in Java segue quanto descritto nelle pubblicazioni del medesimo autore, in quanto la libreria finmath ne propone l'implementazione diretta.

Rispetto al metodo delle differenze finite, l'AD offre numerosi vantaggi. Mentre il primo, la cui definizione è rimandata alle sezioni successive, richiede approssimazioni numeriche che possono introdurre errori significativi e dipende dalla scelta di uno shock ottimale per il calcolo, l'AD fornisce derivate esatte con una precisione analitica. Inoltre, l'AD è spesso più efficiente in termini di tempo computazionale, specialmente in contesti con funzioni di dimensioni elevate e modelli complessi. Questa tecnica trova un vasto impiego in molteplici settori. In finanza quantitativa, ad esempio, è uno strumento utile per il calcolo delle *sensitivities*; nei modelli di *machine learning* è impiegata nell'ottimizzazione delle *loss function*; infine, in ambiti come l'aerospaziale e il meccanico, consente di effettuare analisi di sensitività e ottimizzazione dei progetti con elevata precisione e affidabilità.

Esistono due principali modalità di Differenziazione Automatica: in Avanti (*Forward/Standard/Tangent*) e all'Indietro (*Backward/Adjoint/Reverse*),

Questo capitolo, in particolare, si concentra sulla trattazione delle fondamenta di questa tecnologia, con maggiore enfasi sulla modalità Backward. Nel capitolo successivo, verrà approfondita un'applicazione pratica in Java, nel contesto del calcolo delle sensitivities di un portafoglio equity secondo l'approccio standardizzato (SA) della normativa FRTB, già descritto nel capitolo 1 di questo elaborato, utilizzando quanto già implementato nella libreria finmath.

3.2 L'Automatic Differentiation

Seguendo il filo logico e la notazione riportata in (12), si riporta il primo concetto fondamentale alla base di questa tecnica, ovvero la derivata di una funzione, definita come segue:

Sia $f : \mathbb{R}^m \to \mathbb{R}^n$, $x \mapsto f(x)$, una funzione differenziabile in un punto $x_0 \in \mathbb{R}^m$ se f è definita in quel punto ed esiste una funzione lineare $Df(x_0) : \mathbb{R}^m \to \mathbb{R}^n$ tale che:

$$\lim_{\epsilon \to 0,\, \epsilon \in \mathbb{R}^n} \frac{f(x_0 + \epsilon) - (f(x_0) + Df(x_0)(\epsilon))}{|\epsilon|} = 0. \tag{3.1}$$

La funzione lineare $Df(x_0)$ è chiamata la derivata di f in x_0.

Da questa definizione segue il concetto di derivata parziale: quando una funzione $f : \mathbb{R}^m \to \mathbb{R}^n$ è differenziabile in un punto $x_0 \in \mathbb{R}^m$, è possibile definire le sue derivate parziali $D_i f(x_0) \in \mathbb{R}^m$, per $i = 1, \ldots, m$, come i coefficienti della derivata nella direzione i-esima:

$$D_i f(x_0) = Df(x_0)(e_i), \tag{3.2}$$

dove e_i è il vettore base di $\mathbb{R}^m$ nella i-esima dimensione, ovvero il vettore con tutti i

componenti uguali a 0 eccetto l'elemento nella posizione i, che vale 1. In particolare:

$$D_i f(x_0) = \lim_{\epsilon \to 0} \frac{f(x_0 + \epsilon e_i) - f(x_0)}{\epsilon}. \tag{3.3}$$

Il metodo delle differenze finite è una tecnica per approssimare le derivate parziali di una funzione $f : \mathbb{R}^m \to \mathbb{R}^n$ in un punto $x_0 \in \mathbb{R}^m$. Seguendo la notazione precedente, la derivata parziale nella direzione i-esima può essere calcolata come:

- **Differenza in avanti:**

$$D_i f(x_0) \approx \frac{f(x_0 + \epsilon e_i) - f(x_0)}{\epsilon}. \tag{3.4}$$

- **Differenza all'indietro:**

$$D_i f(x_0) \approx \frac{f(x_0) - f(x_0 - \epsilon e_i)}{\epsilon}. \tag{3.5}$$

- **Differenza centrale:**

$$D_i f(x_0) \approx \frac{f(x_0 + \epsilon e_i) - f(x_0 - \epsilon e_i)}{2\epsilon}. \tag{3.6}$$

La bontà dell'approssimazione tramite differenze finite dipende dalla scelta del parametro ϵ. Infatti, valori troppo piccoli o troppo grandi conducono ad errori di approssimazione.
A differenza del metodo delle differenze finite, l'AD non si basa su approssimazioni numeriche, ma sfrutta la regola della catena per decomporre una funzione complessa in una serie di operazioni fondamentali, ciascuna delle quali può essere facilmente derivata. L'idea centrale è che qualsiasi funzione, indipendentemente dalla sua complessità, può essere vista come una composizione di operazioni aritmetiche di base facilmente derivabili.
La regola della catena, formalmente espressa come:

$$\frac{\partial y}{\partial x_m} = \sum_l \frac{\partial y}{\partial x_l} \cdot \frac{\partial x_l}{\partial x_m}, \tag{3.7}$$

Dove:

- $\frac{\partial y}{\partial x_m}$: Derivata totale di y rispetto a x_m.
- $\frac{\partial y}{\partial x_l}$: Derivata di y rispetto alla variabile intermedia x_l
- $\frac{\partial x_l}{\partial x_m}$: Derivata di x_l rispetto ad x_m.

è il principio cardine su cui si fonda questa metodologia.
Seguendo l'approccio descritto in (9), si può considerare un algoritmo di differenziazione automatica che calcola una funzione

$$y = f(x_0, \ldots, x_{n-1}),$$

in cui sia y sia gli input $x_0, \ldots, x_{n-1}$ sono variabili casuali, con quest'ultime considerate indipendenti.
Per ogni indice m compreso tra n e N vengono definiti dei valori intermedi mediante la relazione

$$x_m = f_m\left(x_{i_1}^{(m)}, \ldots, x_{i_{k(m)}}^{(m)}\right),$$

dove l'operatore f_m agisce su $k(m)$ argomenti individuati dalla lista di indici $\left(i_1^{(m)}, \ldots, i_{k(m)}^{(m)}\right)$.
L'algoritmo consiste in una sequenza ordinata di operazioni, ognuna etichettata da un indice m; il risultato della m-esima operazione, indicato con x_m, viene calcolato in funzione dei risultati ottenuti nelle operazioni precedenti (cioè, dei x_i per $i < m$). In altre parole, ogni passaggio corrisponde all'applicazione dell'operatore f_m sui valori selezionati dagli indici $i_j^{(m)}$.
L'iterazione di questo processo contribuisce alla formazione dell' *operator tree*, in cui i nodi foglia rappresentano le variabili di input $x_0, \ldots, x_{n-1}$.
Gli operatori f_m consistono in operatori elementari per i quali sono note le derivate parziali

$$\frac{\partial f_m}{\partial x_i^{(m)}}.$$

Infine, il calcolo di y (definito come x_n) in funzione degli input $x_0, \ldots, x_{n-1}$ risulta dalla composizione, secondo l'ordine stabilito, degli operatori $f_n, \ldots, f_N$ e dalle rispettive configurazioni degli argomenti.

3.2.1 Forward Automatic Differentiation

La modalità Forward della Differenziazione Automatica è un primo approccio di questo algoritmo per il calcolo delle derivate di una funzione rispetto a una o più variabili indipendenti. Questo metodo sfrutta infatti la regola della catena in modo diretto, propagando le derivate attraverso le operazioni elementari della funzione nella stessa direzione del flusso di calcolo originale (in avanti). In termini pratici, partendo dagli input indipendenti, la modalità forward calcola simultaneamente sia i valori delle funzioni intermedie sia le loro derivate rispetto a una specifica variabile indipendente. Ogni operazione viene differenziata localmente utilizzando le derivate delle operazioni elementari coinvolte. La modalità forward è particolarmente efficiente quando il numero di variabili indipendenti è relativamente piccolo rispetto al numero di variabili dipendenti. Questo la rende ideale per problemi in cui si desidera calcolare la derivata di una funzione rispetto a una singo-

la variabile o a un numero limitato di variabili indipendenti; in questi non vi è infatti la necessità di ripercorrere numerose volte l'intero albero delle operazioni per calcolare la derivata del singolo output.
Riprendendo (9), è possibile formalizzare l'algoritmo forward nel seguente modo: per ogni nodo x_m (con $m = n, n+1, \ldots, N$) e per ciascun argomento indipendente x_i (con $i = 0, \ldots, n-1$), la derivata $\frac{\partial x_m}{\partial x_i}$ si ottiene applicando ripetutamente la regola della catena. Più precisamente, se $x_m = f_m\big(x_{i_1^{(m)}}, \ldots, x_{i_{k(m)}^{(m)}}\big)$ allora:

$$\frac{\partial x_m}{\partial x_i} = \sum_{l=1}^{k(m)} \frac{\partial x_{i_l^{(m)}}}{\partial x_i} \frac{\partial f_m}{\partial x_{i_l^{(m)}}}\big(x_{i_1^{(m)}}, \ldots, x_{i_{k(l)}^{(m)}}\big) \tag{3.8}$$

dove $k(m)$ denota il numero di variabili in ingresso alla funzione f_m e $x_{i_l^{(m)}}$ rappresenta il l-esimo argomento di f_m.
In altre parole, si avanza lungo la sequenza di operazioni (dai nodi iniziali a quelli finali) e, per ogni nodo, si combinano i contributi derivativi dei nodi che lo alimentano, moltiplicandoli per la derivata parziale della funzione che definisce il nodo stesso.
Di seguito verrà presentato un esempio corredato dal relativo albero delle operazioni. Lo stesso esempio sarà poi esaminato in modalità backward per sottolineare le differenze tra le due strategie di calcolo delle derivate.

Esempio 1.1

Consideriamo la funzione:

$$y = \ln\Big(x_1^3 + e^{x_2^2}\Big),$$

e definiamo il valore intermedio:

$$x_4 = x_1^3 + e^{x_2^2}.$$

(I calcoli partono dagli input x_1, x_2 e risalgono l'operator tree fino all'output y):

(1) **Derivate rispetto agli input x_1, x_2:**

$$\frac{\partial x_1}{\partial x_1} = 1, \quad \frac{\partial x_2}{\partial x_2} = 1.$$

(2) **Derivate parziali di x_4 rispetto agli input x_1, x_2:**

$$\frac{\partial x_4}{\partial x_1} = 3x_1^2, \quad \frac{\partial x_4}{\partial x_2} = 2x_2 e^{x_2^2}.$$

(3) **Derivata dell'output rispetto alla variabile intermedia x_4:**

$$\frac{\partial y}{\partial x_4} = \frac{1}{x_4}.$$

(4) **Derivate totali rispetto agli input, applicando la regola della catena:**

$$\frac{\partial y}{\partial x_1} = \frac{\partial y}{\partial x_4} \cdot \frac{\partial x_4}{\partial x_1} = \frac{1}{x_4} \cdot 3x_1^2.$$

$$\frac{\partial y}{\partial x_2} = \frac{\partial y}{\partial x_4} \cdot \frac{\partial x_4}{\partial x_2} = \frac{1}{x_4} \cdot 2x_2 e^{x_2^2}.$$

3.2.2 Backward Automatic Differentiation

Sempre all'interno di (9), viene esposta la modalità backward. Per calcolare la derivata di y rispetto alle variabili indipendenti $x_0, \ldots, x_{n-1}$ in modalità backward, si procede a ritroso lungo il grafo computazionale che descrive il calcolo di y. In pratica, si parte dal nodo finale (che rappresenta appunto y) e si propagano le derivate verso i nodi precedenti, fino a raggiungere le variabili di input, ovvero i nodi foglia dell'operator tree. All'interno di questo grafo, ogni nodo x_l è definito come funzione delle proprie variabili di input, cioè $x_l = f_l\big(x_{i_1^{(l)}}, \ldots, x_{i_k^{(l)}}\big)$. Nella fase di propagazione all'indietro, per ciascun nodo x_m si accumula la derivata parziale $\partial y/\partial x_m$ tramite la regola della catena. Formalmente, se I è l'insieme degli indici l per i quali x_l dipende da x_m, allora:

$$\frac{\partial y}{\partial x_m} = \sum_{l \in I} \frac{\partial y}{\partial x_l} \cdot \frac{\partial f_l}{\partial x_m}\big(x_{i_1^{(l)}}, \ldots, x_{i_{k(l)}^{(l)}}\big) \tag{3.9}$$

dove si assume come condizione iniziale

$$\frac{\partial y}{\partial x_N} = 1 \quad (\text{con } x_N \equiv y).$$

Riassumendo, per ciascun nodo x_l si calcola $\partial y/\partial x_l$ partendo dall'informazione derivativa relativa ai nodi figli, ovvero che dipendono da x_l, moltiplicandola per la derivata locale $\partial f_l/\partial x_m$. Questo processo prosegue in ordine inverso rispetto al flusso del calcolo, consentendo di determinare tutte le derivate $\partial y/\partial x_0, \ldots, \partial y/\partial x_{n-1}$ ripercorrendo all'indietro il grafo una sola volta.
Il vantaggio cruciale di questa metodologia è che si evitano calcoli ridondanti che comparirebbero differenziando ogni operazione in avanti, come spiegato in precedenza. Invece, si registra dapprima la sequenza di operazioni (il "tape") che producono y e, successivamente, si ripercorre l'albero in senso inverso, ricostruendo in modo efficiente i contributi alle derivate.

Di seguito si riporta il corrispettivo backward dell'esempio 1.1 presentato precedentemente:

Esempio 1.2

Utilizziamo la stessa funzione:

$$y = \ln\left(x_1^3 + e^{x_2^2}\right),$$

e definiamo il valore intermedio:

$$x_4 = x_1^3 + e^{x_2^2}.$$

(I calcoli partono dall'output y e si propagano all'indietro verso gli input x_1, x_2):

(1) **Derivata rispetto all'output:**

$$\frac{\partial y}{\partial y} = 1.$$

(2) **Derivata dell'output rispetto alla variabile intermedia x_4:**

$$\frac{\partial y}{\partial x_4} = \frac{1}{x_4} \cdot \frac{\partial y}{\partial y} = \frac{1}{x_4}.$$

(3) **Propagazione verso gli input:** Le derivate di x_4 rispetto agli input sono:

$$\frac{\partial x_4}{\partial x_1} = 3x_1^2, \quad \frac{\partial x_4}{\partial x_2} = 2x_2 e^{x_2^2}.$$

(4) **Derivate totali rispetto agli input, applicando la regola della catena:**

$$\frac{\partial y}{\partial x_1} = \frac{\partial y}{\partial x_4} \cdot \frac{\partial x_4}{\partial x_1} = \frac{1}{x_4} \cdot 3x_1^2.$$

$$\frac{\partial y}{\partial x_2} = \frac{\partial y}{\partial x_4} \cdot \frac{\partial x_4}{\partial x_2} = \frac{1}{x_4} \cdot 2x_2 e^{x_2^2}.$$

In questo caso, la derivata di y rispetto a x_4 viene calcolata una sola volta e utilizzata per propagare tutte le derivate, rendendo l'approccio più efficiente rispetto alla Forward Mode.

Algoritmo backward

Si riporta di seguito la definizione dell'algoritmo backward come definita in (9), questa formulazione differisce lievemente da (3.9), in particolare la sommatoria di quest'ultima viene

sostituita da somme parziali in un ordine diverso. Viene infatti introdotto un meccanismo differente, ovvero di *pushing* delle derivate.
Si definiscono $D_0, \ldots, D_n$ tramite la seguente induzione (all'indietro):

- **Inizializzazione:** $D_n = 1$ e $D_m = 0$ per $m \neq N$.
- **Iterazione:** Procedendo in ordine decrescente lungo la lista degli operatori, per ciascun $m = N, N-1, \ldots, n$ e per ogni argomento $j = 1, \ldots, k^{(m)}$, definiamo:

$$D_{i_j^{(m)}} \to D_{i_j^{(m)}} + D_m \cdot \frac{\partial f_m}{\partial x_{i_j^{(m)}}} \left(x_{i_1^{(m)}}, \ldots, x_{i_k^{(m)}} \right). \tag{3.10}$$

Allora:

$$\forall\, 0 \leq m \leq N : \frac{\partial y}{\partial x_m} = D_m.$$

Come anticipato, la formula utilizza il pushing delle derivate, in cui i contributi vengono "spinti" dai nodi dipendenti x_m verso i nodi argomento $x_{i_j^{(m)}}$, piuttosto che "tirati" (*pulling*) dai nodi dipendenti verso gli argomenti. Questo approccio è computazionalmente più efficiente perché consente un calcolo locale e incrementale delle derivate: ogni nodo x_m aggiorna direttamente le derivate dei suoi argomenti immediati $x_{i_j^{(m)}}$, evitando la necessità di costruire e navigare l'intero grafo computazionale per calcolare le dipendenze globali. Nel pushing, i contributi $D_m \cdot \frac{\partial f_m}{\partial x_{i_j^{(m)}}}$ vengono accumulati istantaneamente nei rispettivi $D_{i_j^{(m)}}$ durante l'iterazione, riducendo il carico computazionale e la memoria richiesta, un vantaggio cruciale per problemi ad alta dimensionalità. Inoltre, il pushing è intrinsecamente più adatto all'implementazione, poiché ogni nodo x_m "spinge" i suoi contributi ai nodi argomento $x_{i_j^{(m)}}$ senza dover sincronizzare l'accumulo dei contributi da altri nodi dipendenti. Questo approccio garantisce quindi un'implementazione più efficiente in termini computazionali rispetto al pulling, sia in termini di tempo che di memoria.

3.2.3 Confronto tra Forward e Backward Differentiation

La propagazione in avanti (Forward Mode) e la propagazione all'indietro (Backward Mode) si differenziano in termini di efficienza computazionale in base al numero di input e output della funzione considerata. Come spiegato da Fries in (9), considerando ad esempio una funzione che mappa uno spazio di dimensione n in uno di dimensione m: $f : \mathbb{R}^n \to \mathbb{R}^m$. Assumendo che il tempo necessario per calcolare le derivate parziali sia paragonabile al tempo richiesto per valutare la funzione stessa, è possibile stimare il tempo totale richiesto per ottenere tutte le derivate di primo ordine.

- **Forward Mode:** in questa modalità, si calcola la derivata rispetto ad una variabile di input per volta, propagandola in avanti attraverso il grafo di calcolo. Di conseguenza, ottenere tutte le derivate rispetto a n variabili indipendenti richiede di ripetere il processo un numero di volte pari a queste ultime, con un costo computazionale che cresce linearmente con n. Se il tempo di valutazione di f è T_f, il costo complessivo risulta:
$$O(n \cdot T_f).$$
Questa strategia risulta particolarmente efficiente quando il numero di input) è molto più piccolo di m (il numero di output), perché richiede un unico passaggio completo per ogni variabile di ingresso da differenziare.

- **Backward Mode:** in questa modalità, si calcolano le derivate di tutti gli input rispetto ad un output per volta, propagando le derivate parziali all'indietro lungo il grafo. Per ricavare le derivate di m output, il procedimento va ripetuto m volte, con un costo di:
$$O(m \cdot T_f).$$
Tale approccio risulta conveniente quando il numero di output è ridotto rispetto a n. In tal caso, ogni passaggio all'indietro fornisce le derivate relative ad uno specifico output, evitando di dover ripetere la computazione per ogni singola variabile di input.

La stessa notazione verrà utilizzata anche nella sezione 3.3.1, in particolare per gli esempi di somma e somma geometrica applicati a simulazioni Monte Carlo.
Nel contesto finanziario, la differenziazione automatica di tipo backward rappresenta uno strumento efficace per il calcolo delle sensitivities di portafogli complessi, specialmente quando vengono utilizzate simulazioni Monte Carlo.
In questi scenari, il portafoglio può essere caratterizzato da un numero elevato di fattori di rischio. Il metodo Monte Carlo, pur essendo flessibile e ampiamente utilizzato per la valutazione di strumenti derivati e portafogli complessi, richiede un elevato sforzo computazionale per il calcolo delle sensibilità con metodi tradizionali, come le differenze finite. Questi approcci, infatti, richiedono la ripetizione di simulazioni complete per ogni fattore di rischio, perturbando ciascun parametro di input in modo indipendente. Questo porta a una crescita lineare del costo computazionale con il numero di fattori di rischio. Anche con la modalità forward dell'AD vi è la necessità di ripercorrere più volte gli stessi paths. Al contrario, l'Automatic Adjoint Differentiation (AAD) sfrutta la modalità backward, consentendo di calcolare tutte le sensibilità rilevanti in un singolo passaggio all'indietro.
Grazie a questa tecnica, l'AAD permette di ottenere le derivate con un'efficienza computazionale notevolmente superiore, riducendo drasticamente i tempi di calcolo e ottimizzando l'utilizzo delle risorse hardware. Un utile, nonché recente, riferimento per approfondire le differenze tra i due metodi è rappresentato da (14).

3.2.4 Stochastic Automatic Differentiation

Una variante dell'Automatic Differentiation è quella applicata a contesti stocastici, in cui entrano in gioco le variabili casuali e gli operatori che le trasformano. In questi contesti, le quantità di interesse non sono più semplici valori scalari, ma vettori e matrici che rappresentano traiettorie di processi stocastici, appunto, come quelle generate da una simulazione Monte Carlo. Questo implica un cambiamento fondamentale nella struttura delle derivate e nella gestione delle informazioni, appesantendo così l'utilizzo di memoria utilizzata per ripercorrere l'operator tree.
L'approccio si estende quindi al concetto di *Stochastic Automatic Differentiation* (in questo elaborato non verrà adottata una nomenclatura distinta per indicare l'AAD in contesti deterministici o stocastici), che applica le tecniche di differenziazione automatica direttamente alle variabili casuali. In particolare, quando un operatore agisce su una variabile aleatoria, il risultato è rappresentabile in forma vettoriale o matriciale, poiché coinvolge più realizzazioni dello stesso processo.
Per comprendere le implicazioni di questa estensione, è utile distinguere tra operatori *path-wise* e operatori *non path-wise*. Dato il contesto di una simulazione Monte Carlo possiamo dire che i primi, di cui fa parte ad esempio la somma tra due variabili casuali, operano separatamente su ciascun percorso simulato, applicando la trasformazione in modo indipendente per ogni traiettoria del processo stocastico. Questa computazione "locale" semplifica il calcolo delle derivate, rendendolo più efficiente e parallelizzabile, poiché ogni realizzazione del processo può essere trattata in maniera isolata. Al contrario, gli operatori non path-wise aggregano informazioni provenienti da molteplici percorsi della simulazione, introducendo una dipendenza tra traiettorie diverse. Un esempio tipico è l'operatore di aspettativa condizionale, che calcola il valore atteso di una variabile data un certo insieme di informazioni; ad esempio, esso è di fondamentale importanza per il pricing delle opzioni Bermuda. In questi casi, il calcolo delle derivate diventa più complesso, poiché non è possibile differenziare percorso per percorso, e occorre tenere traccia delle dipendenze tra le realizzazioni. In (9) viene proposto un metodo per la gestione di tali operatori, che permette un considerevole risparmio computazionale.
In questa sezione, verrà illustrato come queste tecniche possono essere utilizzate per calcolare il valore atteso e il valore atteso condizionale, mentre non verrà approfondito il trattamento delle funzioni indicatrici. Tale scelta è coerente con la tipologia di strumenti utilizzati nel capitolo 3 di questo elaborato, ovvero opzioni europee e di tipo bermuda.

Implementazione dell'operatore aspettativa

Il valore atteso è uno strumento fondamentale nel calcolo stocastico, particolarmente rilevante nel pricing di strumenti finanziari. Rappresenta la media ponderata delle realizzazioni di una variabile casuale, dove i pesi sono determinati dalla probabilità associata a ciascun esito. Nel contesto delle simulazioni Monte Carlo, il valore atteso si calcola utiliz-

zando i risultati generati da n path simulati, ciascuno rappresentante un possibile scenario di evoluzione dei fattori di mercato. Formalmente, seguendo (9), il valore atteso è stimato come la media dei payoff associati ai n campioni $Y(\omega_k)$, ed è espresso come:

$$Z = \mathbb{E}(Y) := \frac{1}{n} \sum_{k=1}^{n} Y(\omega_k). \tag{3.11}$$

Come accennato in precedenza, gli operatori non path-wise differiscono da quelli path-wise per il modo in cui elaborano le informazioni della simulazione: mentre questi ultimi trasformano ciascun percorso simulato in modo indipendente, i primi, come il valore atteso, combinano i risultati di molteplici paths. Questa proprietà implica che la derivata del valore atteso rispetto al vettore Y, che rappresenta le realizzazioni corrispondenti ai vari paths, genera una matrice completa, in cui ogni elemento dipende da tutti i paths simulati. Formalmente, questa derivata è data da:

$$\frac{\partial Z}{\partial Y} = \frac{1}{n} \begin{pmatrix} 1 & 1 & \cdots & 1 \\ 1 & 1 & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \cdots & 1 \end{pmatrix}.$$

Se consideriamo Y come una funzione di X, ossia $Y = Y(X)$, allora per $Z = \mathbb{E}(Y)$, la derivata rispetto a X si scrive come:

$$\frac{\partial Z}{\partial X} = \frac{1}{n} \begin{pmatrix} 1 & 1 & \cdots & 1 \\ 1 & 1 & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \cdots & 1 \end{pmatrix} \cdot \frac{\partial Y}{\partial X}(\omega_k).$$

Un vantaggio fondamentale del valore atteso è la sua linearità. Se $Z = f(Y)$ e f è un operatore path-wise, allora la derivata del valore atteso può essere espressa come:

$$\frac{\partial}{\partial Y}\mathbb{E}[Z] = \mathbb{E}\left[\frac{\partial Z}{\partial Y}\right]. \tag{3.12}$$

Questa proprietà consente di spostare l'operatore di differenziazione all'interno del valore atteso, permettendo di esprimere la derivata come la media delle derivate dei singoli path. Questo riduce significativamente il costo computazionale associato alla propagazione delle derivate lungo l'operator tree. Tuttavia, nel contesto dell'AAD, l'introduzione di operatori non path-wise come il valore atteso introduce alcune sfide significative.
Per affrontare la complessità computazionale associata alla derivata del valore atteso, sono state proposte due metodologie principali. La prima consiste nel modificare l'algorit-

mo AAD per spezzare l'operator tree ogni volta che si incontra un valore atteso. Questo approccio segmenta il calcolo delle derivate in sotto-strutture indipendenti, riducendo la propagazione della matrice completa. Alla luce di questo, considerando la notazione utilizzata finora e definendo l'operatore aspettativa come x_k, ovvero $x_k = f_k(x_1^{(k)}) = \mathbb{E}(x_1^{(k)})$, si ottiene:

$$\frac{\partial y}{\partial x_m} = \frac{\partial y}{\partial x_k}\bigg|_{x_k} + \frac{\partial y}{\partial x_k}\bigg|_{x_k} \cdot \mathbb{E}\left(\frac{\partial x_{1^{(k)}}}{\partial x_m}\right) \tag{3.13}$$

con

$$\frac{\partial y}{\partial x_m}\bigg|_{x_k} = \sum_{l \in I, l \neq k} \frac{\partial y}{\partial x_l}\bigg|_{x_k} \cdot \frac{\partial f_l}{\partial x_m}(x_{i_1^{(l)}}, \ldots, x_{i_{k^{(l)}}^{(l)}})$$

per $m = N-1, N-2, \ldots, n$

e

$$\frac{\partial x_1^{(k)}}{\partial x_m} = \sum_{l \in I} \frac{\partial x_1^{(k)}}{\partial x_l} \cdot \frac{\partial f_l}{\partial x_m}(x_{i_1^{(l)}}, \ldots, x_{i_{k^{(l)}}^{(l)}})$$

per $m = i_1^{(k)} - 1, i_1^{(k)} - 2, \ldots, n$

e dove $\frac{\partial x_1^{(k)}}{\partial x_m} = 0$ per $m > i_1^{(k)}$.

Tuttavia, come sottolineato da Fries, *"this destroys the elegance of the backward automatic differentiation"*. Questo avviene perché il calcolo della derivata $\frac{\partial x_1^{(k)}}{\partial x_m}$ richiede un primo passaggio separato, e il risultato viene trattato come un nodo foglia (*leaf-node*) nell'albero di calcolo. In un secondo passaggio, l'operatore valore atteso viene applicato al risultato ottenuto. Questo approccio introduce una significativa complessità nel processo computazionale, riducendo l'efficienza dell'AAD.

La seconda metodologia, invece, sfrutta l'assunzione che il valore atteso sia l'ultimo operatore nell'operator tree (f_N). Questa ipotesi, coerente con il pricing di molti strumenti finanziari con simulazioni Monte Carlo, consente di calcolare le derivate in modo più efficiente evitando la propagazione esplicita della matrice completa. La formula (3.13) diventa quindi:

$$\frac{\partial y}{\partial x_m} \overset{\mathbb{E}}{=} \frac{\partial y}{\partial x_k}\bigg|_{x_k} + \mathbb{E}\left(\frac{\partial y}{\partial x_k}\bigg|_{x_k}\right) \cdot \frac{\partial x_1^{(k)}}{\partial x_m}, \tag{3.14}$$

Grazie a questa assunzione, infatti, è possibile modificare l'algoritmo citato in (3.10) come descritto dal seguente teorema, introdotto da Fries:

- **Inizializzazione:**

$$D_N^{\mathcal{G}} = 1 \quad \text{e} \quad D_m^{\mathcal{G}} = 0 \quad \text{per ogni } m \neq N.$$

- **Iterazione *backward*:** Per ogni $m = N, N-1, \ldots, n$ (Iterando all'indietro lungo la lista degli operatori), si definisce:

$$D_{i_j^{(m)}}^{\mathcal{G}} \to \begin{cases} D_{i_j^{(m)}}^{\mathcal{G}} + D_m^{\mathcal{G}} \cdot \frac{\partial f_m}{\partial x_{i_j^{(m)}}} \left(x_{i_1^{(m)}}, \ldots, x_{i_{k^{(m)}}^{(m)}} \right) & \text{se } f_m \notin \mathcal{G}, \\ D_{i_j^{(m)}}^{\mathcal{G}} + G^* \left(D_m^{\mathcal{G}} \right) & \text{se } f_m = G \in \mathcal{G}. \end{cases}$$

con

- $\mathcal{G}$ una famiglia di operatori lineari tale che, per ogni variabile casuale A, B e per ogni $G \in \mathcal{G}$, vale:

$$\exists G^* : \mathbb{E}(A \cdot G(B)) = \mathbb{E}(G^*(A) \cdot B).$$

- f_m sufficientemente regolari affinché:

$$\frac{d}{dx_j} G(f_m) = G\left(\frac{df_m}{dx_j} \right).$$

Quanto appena descritto fa riferimento al teorema 1 di (9). All'interno del paper è possibile consultare anche la relativa dimostrazione.

Nel contesto del pricing di strumenti finanziari come le opzioni Bermuda, il valore atteso condizionato gioca un ruolo fondamentale nel calcolo dei payoff ottimali nei momenti intermedi di esercizio, permettendo di considerare scenari futuri basati sulle informazioni attualmente disponibili. Una proprietà chiave del valore atteso condizionato, ripresa dal paper, è la *tower law*, che afferma:

$$\mathbb{E}(A \cdot \mathbb{E}[B|\mathcal{F}_t]) = \mathbb{E}(\mathbb{E}[A|\mathcal{F}_t] \cdot B) = \mathbb{E}(\mathbb{E}[A \cdot B|\mathcal{F}_t]), \tag{3.15}$$

dove A e B sono variabili casuali. Questa legge evidenzia come il valore atteso condizionato preservi le proprietà del valore atteso totale e renda possibile un'efficiente separazione delle informazioni temporali nel calcolo delle derivate.
Un punto cruciale evidenziato da Fries è che la filtrazione $\mathcal{F}_t$ non dipende direttamente dalle variabili indipendenti del modello in applicazioni pratiche come quelle che coinvolgono simulazioni Monte Carlo. Questa assunzione elimina la necessità di differenziare

la filtrazione stessa. Inoltre, un altro aspetto chiave discusso nel paper è l'importanza di evitare di derivare direttamente l'approssimazione del valore atteso condizionato (*American Monte-Carlo*), passaggio che può indurre errori di approssimazione. Differenziare un'approssimazione, come quella ottenuta tramite regressione lineare, può portare infatti ad una propagazione dell'errore.
Una metodologia particolarmente efficace per approssimare il valore atteso condizionato è stata introdotta da Longstaff e Schwartz (13) come *Least Squares Monte Carlo* (LSMC). In questo approccio, il valore atteso condizionato viene stimato rappresentando la variabile casuale di interesse V come una combinazione lineare di un insieme di *basis functions* X, che dipendono da un vettore di variabili casuali Z che genera la filtrazione $\mathcal{F}_t$. Formulando il problema come una minimizzazione, il valore atteso condizionato è definito come:

$$\mathbb{E}[V|\mathcal{F}_t] = \mathbb{E}[V|Z] = \underset{g\in\{f(Z)\}}{\operatorname{argmin}} \|V - g\|_{L_2}. \tag{3.16}$$

Nella pratica, lo spazio $\{f(Z)\}$ viene approssimato utilizzando una combinazione lineare di funzioni base X:

$$\mathbb{E}[V|Z] \approx \underset{g\in\{X\cdot\alpha|\alpha\in\mathbb{R}^m\}}{\operatorname{argmin}} \|V - g\|_{L_2}.$$

La soluzione a questo problema di minimizzazione è ottenuta tramite la regressione lineare, dove i coefficienti α sono calcolati come:

$$\alpha = (X^\top X)^{-1} X^\top V,$$

e il valore atteso condizionato approssimato si scrive:

$$\mathbb{E}[V|Z] \approx X \cdot \alpha.$$

Qui, X è una matrice le cui colonne rappresentano i valori delle funzioni base nei vari path Monte Carlo, e V è il vettore dei payoff simulati.

3.3 Implementazione all'interno della libreria finmath in Java

L'implementazione di Automatic Differentiation (AD) con un linguaggio orientato agli oggetti ha solide fondamenta teoriche grazie a numerosi contributi della letteratura. Ad esempio, il lavoro di Capriotti e Giles (5) fornisce una base metodologica, includendo rappresentazioni in pseudo-codice che illustrano le linee guida per l'implementazione degli algoritmi di differenziazione automatica. Parallelamente, Henrard (12) esplora l'applicazione pratica in contesti finanziari dell'AD, partendo dal concetto chiave di composizione

delle funzioni. Questo approccio viene adattato a strutture algoritmiche di base, accompagnate da funzioni di esempio, fino a trattare casi concreti come l'applicazione della formula di Black e del modello SABR, dimostrando l'efficacia di tali metodi nell'implementazione dei modelli utilizzati in ambito finanziario. Dando per scontata la base della programmazione ad oggetti, si vogliono ora riportare quelli che sono i concetti chiave che permettono di implementare in modo efficace l'AD in Java, soprattutto per quanto riguarda la modalità adjoint, ottimizzando le tempistiche computazionali e i requisiti di memoria.

3.3.1 Concetti chiave

Un primo concetto chiave è quello di oggetto immutabile, il quale, una volta creato, non può essere modificato. Questo approccio è cruciale nell'implementazione di AD, soprattutto per la rappresentazione delle variabili casuali, dove è necessario ottimizzare la gestione della memoria e garantire consistenza nei calcoli. Invece di creare copie multiple, si memorizzano solo riferimenti agli oggetti immutabili, migliorando l'utilizzo della memoria. L'immutabilità consente quindi di salvare una rappresentazione unica e stabile dei nodi dell'operator tree, migliorando l'efficienza del processo di backward differentiation e supportando la *lazy evaluation*.
Quest'ultima è infatti un altro elemento fondamentale, grazie ad essa è possibile rinviare il calcolo dei valori fino a quando non sono effettivamente necessari. Questo principio consente di costruire il grafo computazionale in modo incrementale, registrando le operazioni senza eseguirle immediatamente. Quando viene richiesto un valore o una derivata, il sistema attraversa il grafo e calcola solo le quantità rilevanti, ottimizzando l'uso delle risorse computazionali. Nel caso della modalità backward, un aspetto critico è la gestione dell'operator tree. Ogni nodo di questo albero contiene informazioni sull'operazione e sugli argomenti coinvolti, che vengono utilizzate nel momento della differenziazione. La memorizzazione di questa struttura può comportare un elevato consumo di memoria, specialmente quando ad essere implementati sono modelli complessi. In (9), Fries presenta due esempi significativi per illustrare i concetti di oggetti immutabili e lazy evaluation: la funzione somma e somma geometrica. Questi esempi forniscono un esempio esplicativo di come tali tecniche possano rafforzare l'efficienza e la gestione della memoria nell'implementazione dell'AAD.

(1) **Somma**: si considera la funzione $\text{sum}_{i+1} = f(\text{sum}_i, x_i) = \text{sum}_i + x_i$, dove con x_i si intendono delle variabili casuali, il codice registra ogni operazione come un nodo nell'operator tree. Ognuno dei quali memorizza il riferimento alla funzione `sum`, che rappresenta appunto un nodo dell'albero. Non memorizza invece i riferimenti agli argomenti (sum_i, x_i). In questo caso si deduce dall'operatore `sum` che i riferimenti agli argomenti non sono necessari e dunque possono non essere memorizzati, dato che la derivata parziale della somma è sempre 1. Considerando una simulazione Monte Carlo, dove N è il numero di derivate parziali e m il numero di paths, la

memoria richiesta è ridotta a $O(N)$, a differenza dell'approccio tradizionale che richiederebbe $O(N + m)$, memorizzando tutti le derivate parziali.

(2) **Somma Geometrica**: si considera poi la funzione $\text{sum}_{i+1} = f(\text{sum}_i, x_i) = \text{sum}_i + x_i$, con $x_i = g(x, i) = x^i$. Dove per il calcolo di $x_i = g(x, i) = x^i$, l'implementazione memorizza: un riferimento all'oggetto immutabile x e all'indice i, che definisce l'esponente. Invece di salvare tutte le variabili intermedie, come $i \cdot x^{i-1}$, l'implementazione registra il riferimento all'operazione di elevamento a potenza (`pow` e agli argomenti (x, i), dove x è un'istanza dell'oggetto che rappresenta x. La derivata parziale $\partial g(x, i)/\partial x = i \cdot x^{i-1}$ viene calcolata solo al momento del backward pass, grazie all'uso della lazy evaluation, che permette di attraversare l'albero degli operatori e calcolare le sensibilità solo quando necessario. Questo approccio riduce ulteriormente la memoria necessaria, dato che il valore di x viene memorizzato una sola volta. Il requisito di memoria in questo diventa $O(N) + O(m)$.

Un altro elemento rilevante nell'implementazione è il trattamento degli operatori non path-wise, come il valore atteso e il valore atteso condizionale. Finmath gestisce questi operatori utilizzando diverse tecniche di approssimazione, come quella citata alla fine della sezione (3.2.4). Al pari degli altri operatori come somma, sottrazione ecc, anche l'aspettativa condizionale gode di un metodo ad-hoc che ne consente il corretto utilizzo. Nel calcolo dell'aspettativa condizionale, è cruciale tracciare il tempo di filtrazione associato a ogni variabile casuale, cioè l'istante a partire dal quale l'informazione risulta disponibile. Per stabilire tale istante in corrispondenza di un'operazione, si adotta il valore massimo tra i tempi di filtrazione dei suoi argomenti. Alle costanti si assegna convenzionalmente un tempo di filtrazione pari a $-\infty$, a indicare che non dipendono da alcun processo stocastico. Al contrario, un incremento Browniano $W(t + \Delta t) - W(t)$ ha come tempo di filtrazione $t + \Delta t$, rappresentando il momento finale dell'intervallo su cui è calcolato l'incremento. Questo monitoraggio continuo dell'informazione consente di introdurre un controllo preliminare, in grado di determinare se una particolare aspettativa condizionata debba effettivamente essere calcolata. Qualora il controllo indichi che l'operazione non risulti necessaria, si evita il calcolo, riducendo così il carico computazionale e il consumo di memoria. In tal modo, si enfatizza ancora una volta l'importanza di escludere le elaborazioni superflue per migliorare l'efficienza dell'algoritmo.

3.3.2 Classi principali

Vista la maggiore aderenza dell'Automatic Differentiation di tipo backward a calcoli con molti input e pochi output, come il pricing di strumenti finanziari mediante simulazioni Monte Carlo, in questa sezione si tratteranno le caratteristiche di questa sola metodologia, mentre non verrà discussa quella di tipo forward.

Il cardine dell'implementazione è definito dall'interfaccia di finmath `RandomVariableDifferentiableAAD`, che a sua volta implementa `RandomVariableDifferentiable`.Quest'ultima estende `RandomVarible` implementando dei metodi per gestire la differenziazione automatica, ovvero:

- `getID()`: metodo per recuperare l'ID identificativo di un nodo dell'operator tree.
- `getGradient()`: metodo per svolgere la differenziazione automatica di tipo backward.
- `getTangent()`: metodo per svolgere la differenziazione automatica di tipo forward.

Di seguito vengono riportate le caratteristiche principali di questa interfaccia e ne viene brevemente riassunta la logica di funzionamento.

Il primo elemento fondamentale della classe riguarda la costruzione e la gestione dell'operator tree, che rappresenta la struttura computazionale delle operazioni matematiche applicate alle variabili differenziabili. La sua implementazione è affidata alla classe interna statica `OperatorTreeNode`, il cui scopo è modellare ogni operazione come un nodo di un grafo computazionale orientato.
Ogni nodo `OperatorTreeNode` è caratterizzato dai seguenti elementi fondamentali:

- Tipo di operazione (`OperatorType`): definito tramite una `enum`, rappresenta le operazioni matematiche fondamentali supportate, come somma (`ADD`), moltiplicazione (`MULT`), divisione (`DIV`) e altre operazioni comuni nell'analisi differenziale automatica.
- Identificativo univoco (`ID`): ogni nodo è associato a un valore univoco generato dinamicamente.
- Riferimenti ai nodi dipendenti (`arguments`): ogni nodo mantiene un riferimento ai propri argomenti, ossia agli altri `OperatorTreeNode` che ne costituiscono gli operandi. Questo permette di rappresentare in modo esplicito la dipendenza tra le operazioni e di costruire l'albero computazionale, necessario per il backward pass nell'algoritmo AAD.
- Valori numerici degli argomenti (`argumentValues`): per ottimizzare la gestione della memoria, la classe mantiene esplicitamente i valori numerici degli operandi solo nei casi in cui siano effettivamente necessari. Ad esempio, per operazioni come `ADD` o `SUB`, il valore dei singoli argomenti non è rilevante nella fase di differenziazione e viene quindi eliminato, riducendo il tempo di computazione e migliorando l'efficienza della memoria.

Il metodo `getPartialDerivative` calcola la derivata parziale dell'operatore corrente (nodo) rispetto a uno dei suoi argomenti (nodi), identificato tramite il parametro `differentialIndex`. Dopo aver verificato che l'argomento è effettivamente presente nella lista `arguments`, il metodo estrae i valori numerici associati agli argomenti del nodo (ad esempio, X, Y, Z, se disponibili) e determina la derivata in base al tipo di operatore (`operatorType`). Per operatori come `EXP`, `LOG`, e `SQRT`, la derivata è calcolata applicando le regole analitiche standard di differenziazione. Nel caso di operatori binari, come `MULT`, `DIV`, `ADD` e `SUB`, la derivata dipende dalla posizione dell'argomento all'interno dell'operazione, distinguendo il contributo di ciascun operando nel rispetto della regola del prodotto e della regola del quoziente. Se l'operatore corrente è di natura statistica, come `VARIANCE` o `STDEV`, il metodo impiega le relative formule statistiche per calcolare le derivate parziali, tenendo conto della struttura campionaria dell'operatore. Gli operatori quali `MIN` e `MAX` utilizzano una funzione indicatrice che assume valore 1 per gli elementi che realizzano rispettivamente il minimo o il massimo, e 0 negli altri casi, rendendo il calcolo delle derivate sensibile ai valori dei dati in input. Un caso particolare è rappresentato dall'operatore `CHOOSE`, che corrisponde a una funzione indicatrice. Per questo operatore, la derivata viene approssimata in base al metodo scelto per la gestione della *Dirac Delta*. Per una trattazione più approfondita di questa approssimazione e delle sue implicazioni nel contesto della differenziazione automatica stocastica, si rimanda a (10). Il risultato finale del metodo è un'istanza di `RandomVariable` che rappresenta la derivata parziale dell'operazione rispetto all'argomento specificato. Questo valore viene poi utilizzato nel calcolo della derivata totale lungo il grafo computazionale, garantendo una corretta propagazione dei gradienti nel framework AAD.

Successivamente, il metodo `propagateDerivativesFromResultToArgument` gestisce la propagazione delle derivate lungo l'operator tree, applicando la regola della catena per aggiornare i contributi delle variabili argomento. Questo metodo costituisce il nucleo dell'implementazione dell'algoritmo AAD, come descritto in 3.10. Per ciascun nodo del grafo computazionale, la derivata parziale dell'operazione corrente rispetto a ciascun argomento (`partialDerivative`) viene moltiplicata per la derivata totale già calcolata del nodo stesso (`derivative`). In questo modo, il contributo di ogni nodo viene propagato lungo la struttura dell'operatore in modo efficiente. Se il nodo rappresenta un operatore stocastico, come `AVERAGE`, `CONDITIONAL EXPECTATION` o `CHOOSE`, è necessario applicare un trattamento specifico per garantire un'adeguata gestione della propagazione delle derivate. In particolare, nel caso dell'operatore di valore atteso condizionale, viene utilizzato un oggetto `ConditionalExpectationEstimator` per stimare l'aspettativa condizionata di `derivative` rispetto alla filtrazione $\mathcal{F}_t$. Un'implementazione tipica di questo approccio prevede l'uso di un `MonteCarloConditionalExpectationRegression` come `estimator`, permettendo di realizzare l'approssimazione definita in 3.2.4. Questo approccio consente di evitare la differenziazione diretta di un'approssimazione, riducendo

significativamente il rischio di propagazione degli errori. Se la derivata dell'argomento non è ancora stata inizializzata, viene assegnato il valore del prodotto `derivative` × `partialDerivative`. In caso contrario, la derivata viene aggiornata in modo incrementale attraverso il metodo `addProduct`, che somma il nuovo contributo a quello esistente. Infine, il valore aggiornato della derivata viene memorizzato all'interno della mappa `derivatives`, garantendo un'efficiente propagazione dei gradienti lungo il grafo computazionale e ottimizzando la gestione della memoria e delle risorse computazionali.

Infine, il metodo `getGradient` implementa il backward pass di AAD, calcolando il gradiente della variabile differenziabile corrente rispetto a un insieme di variabili indipendenti. Per garantire un'elaborazione corretta e ordinata dei nodi, utilizza una struttura dati `TreeMap`, che organizza i nodi in ordine decrescente rispetto al loro `ID`. Questa scelta assicura che ogni nodo venga processato prima di propagare la propria derivata ai suoi argomenti, rispettando la struttura del grafo computazionale. Il ciclo principale del metodo estrae iterativamente il nodo con l'identificativo massimo e, per ciascun argomento del nodo, calcola la derivata parziale tramite il metodo `propagateDerivativesFromResultToArgument`. Questo processo consente di aggiornare la mappa `derivatives` applicando in modo sistematico la regola della catena, garantendo la corretta propagazione dei gradienti lungo l'operator tree. Se la configurazione del gradiente prevede la memorizzazione esclusiva delle derivate relative ai nodi foglia — ossia le variabili indipendenti del grafo — i nodi intermedi vengono rimossi dalla mappa `derivatives` non appena i loro contributi sono stati completamente propagati ai rispettivi argomenti. Questo comportamento è regolato dal metodo `isGradientRetainsLeafNodesOnly`, che stabilisce se l'output del gradiente debba contenere esclusivamente le derivate finali necessarie per il calcolo delle sensibilità, eliminando i risultati intermedi non più utili. Infine, prima di restituire il risultato finale, il metodo esegue un'operazione di pulizia sulla mappa `derivatives`, rimuovendo tutti i nodi presenti nell'insieme `independentIDs`. Questo insieme raccoglie gli `ID` delle variabili indipendenti definite dall'utente e viene utilizzato per evitare ridondanze nel risultato, assicurando che il gradiente finale contenga solo le informazioni essenziali richieste dall'analisi delle sensibilità.

I metodi analizzati rappresentano gli elementi essenziali per l'implementazione dell'Adjoint Automatic Differentiation e per la gestione efficiente delle derivate lungo l'operator tree, assicurando che il calcolo dei gradienti avvenga in modo ottimizzato e scalabile. In particolare, l'adozione di una struttura basata su un consente di organizzare le operazioni in maniera strutturata, migliorando sia l'efficienza computazionale che la tracciabilità delle dipendenze tra le variabili.
La scelta di rappresentare il flusso computazionale attraverso un DAG si rivela fondamentale, poiché consente di modellare in modo esplicito le dipendenze tra le operazioni

e garantisce che il calcolo delle derivate possa essere eseguito in modo sistematico ed efficiente. Ogni nodo del grafo rappresenta un'operazione matematica, mentre gli archi definiscono le relazioni di dipendenza tra gli operatori, assicurando che il flusso dei calcoli segua una direzione ben definita senza possibilità di cicli. L'adozione di questa struttura porta con sé diversi vantaggi dal punto di vista computazionale. Dal punto di vista dell'analisi e del debugging, il DAG offre un modello chiaro e tracciabile del calcolo, facilitando l'identificazione di errori e la comprensione delle relazioni tra le variabili differenziabili.
Oltre ai metodi trattati, l'interfaccia include numerosi strumenti che contribuiscono alla flessibilità e modularità dell'implementazione. Proprio quest'ultima ne garantisce poi la facile estensione. Un aspetto rilevante è costituito dagli `@override` dei metodi definiti nell'interfaccia `RandomVariable`, i quali estendono e personalizzano il comportamento di quest'ultima per adattarsi alle esigenze dell'algoritmo AAD.
Il prossimo step, che costituisce l'ultimo capitolo di questo elaborato, consiste nell'utilizzo di questa interfaccia al fine di calcolare le sensitivities di un portafoglio equity, nell'ambito del sensitivities-based method della normativa FRTB descritto in 2. L'obiettivo è dimostrare come sia possibile sostituire il metodo delle differenze finite, come previsto dalla normativa, con uno più preciso ed efficiente quale è l'AAD.

4 Implementazione in Java

4.1 Introduzione al progetto Java

Questo capitolo illustra l'implementazione in Java dei concetti sviluppati nei capitoli precedenti, con particolare attenzione all'uso della libreria finmath. La scelta di questa libreria non è casuale, ma coerente con i lavori di Fries già citati in precedenza, rappresentando uno standard consolidato per il calcolo delle sensibilità attraverso l'Automatic Adjoint Differentiation (AAD). L'obiettivo principale dell'implementazione è integrare l'algoritmo AAD nel calcolo delle sensitivities di un portafoglio composto da strumenti equity, analizzando il suo impatto in termini di precisione ed efficienza computazionale rispetto al metodo delle differenze finite. Quest'ultimo, come noto, fornisce un'approssimazione della derivata di una funzione e richiede una ricalibrazione iterativa per ottenere risultati affidabili, con un conseguente incremento del costo computazionale. L'uso di AAD, invece, consente di calcolare la derivata in modo esatto e analitico, riducendo significativamente sia l'errore numerico sia il tempo di elaborazione.
Per mantenere un livello di coerenza con la parte teorica descritta nel capitolo precedente e per non elevare oltremisura la complessità dell'implementazione, il portafoglio scelto sarà composto da strumenti finanziari relativamente semplici ma rappresentativi delle principali categorie di rischio trattate. In particolare, saranno considerate opzioni europee, opzioni bermuda e azioni, con l'obiettivo di calcolare Delta, Vega e Curvature Risk. La scelta delle opzioni europee risponde all'esigenza di avere un primo esempio di implementazione relativamente semplice, che consente anche un confronto diretto con le formule analitiche del modello di Black-Scholes. L'inclusione delle opzioni bermuda introduce invece un ulteriore livello di complessità, poiché il loro pricing richiede l'utilizzo di approssimazioni del valore atteso condizionale, un aspetto già discusso nel capitolo precedente e che risulta fondamentale nell'applicazione di AAD in contesti più realistici. Infine, l'inserimento delle stock riflette la loro importanza nei portafogli di trading delle banche, poiché notevolmente presenti. Tuttavia, a differenza delle opzioni, il calcolo delle sensitivities per le azioni non richiede l'uso di AAD, poiché il delta è semplicemente pari al prezzo corrente dell'azione, mentre il vega è nullo.
L'implementazione non si limita alla valutazione delle sensibilità, ma si inserisce in un framework più ampio che simula il processo di aggregazione previsto dalla normativa FRTB. Oltre al calcolo delle sensibilità nette, il codice è stato strutturato per eseguire l'aggregazione intra-bucket e inter-bucket, seguendo le regole dettate dal framework regolamentare.

In questo modo è possibile determinare il requisito di capitale regolamentare in modo dinamico, sfruttando coefficienti di ponderazione del rischio e parametri di correlazione specifici per ciascun bucket. Per garantire un maggiore realismo e flessibilità, il codice elabora i dati di input attraverso file CSV, evitando l'hard coding e permettendo una gestione scalabile delle informazioni. Il file principale contiene i trades del portafoglio, strutturato sulla falsa riga del formato ISDA® FRTB-SA CRIF, così da garantire un'organizzazione coerente con gli standard di settore. Insieme a questo, è previsto un ulteriore file di input che permette di gestire il mapping tra underlying e bucket di rischio, consentendo di associare ogni fattore di rischio al relativo bucket senza che questa relazione sia impressa nel codice.
Sebbene l'uso di AAD migliori la precisione del calcolo delle sensibilità, è importante considerare che il metodo non elimina completamente gli errori di approssimazione, in particolare quelli legati all'uso delle simulazioni Monte Carlo. La convergenza delle simulazioni e la scelta del numero di percorsi influenzano direttamente la qualità dei risultati e rappresentano un compromesso tra accuratezza e costo computazionale.

Nel corso del capitolo verrà analizzato nel dettaglio il contesto applicativo e le metodologie adottate, con un approfondimento sulla struttura del progetto Java, sulla logica di calcolo delle sensibilità e sulla loro aggregazione. Verrà inoltre spiegato il funzionamento del codice, illustrando il flusso operativo dalla lettura dei dati all'output finale. Alla fine del capitolo sarà infine possibile consultare il codice completo dell'implementazione.

4.2 Analisi del codice

Di seguito si riporta un'analisi della struttura del codice, con riferimenti alle metodologie teoriche adottate, al fine di garantire una chiarezza di lettura; tale descrizione seguirà un approccio diviso per classi.

CSVParser

Come accennato nell'introduzione, il punto di partenza del progetto è rappresentato dal portafoglio contenente diversi trades caratterizzati da strumenti finanziari differenti. Al fine di rendere l'input modificabile, in particolare nel numero di strumenti considerati, si utilizza un file CSV `sample_trades.csv` contenuto nelle risorse del progetto. La struttura di questo file è standard e riprende le informazioni chiave di ogni posizione assunta nel desk fittizio. Le tipologie di strumenti considerate sono solo tre (opzioni europee, opzioni bermuda e azioni), mentre le caratteristiche di ciascuno di questi strumenti sono variabili. Si ricorda che, per l'implementazione, è stato considerato come unico risk factor: il prezzo spot del sottostante per il calcolo del delta e del curvature risk, mentre per il

vega le volatilità implicite mappate nei *tenors* regolamentari. Non sono stati inclusi i tassi repo, sebbene la struttura del file sia predisposta per l'integrazione di tali componenti in eventuali sviluppi futuri.

La classe `CsvParser` si occupa della lettura e dell'interpretazione di questo file CSV utilizzando la libreria *Apache Commons CSV*, che consente un parsing [1] affidabile ed efficiente. Il metodo `parseCsvTrade()` converte ogni riga del CSV in un oggetto `Trade`, garantendo che i dati siano interpretati correttamente nei loro tipi appropriati. Durante questa fase, vengono eseguiti controlli per prevenire errori di parsing e vengono assegnati valori di default in caso di campi mancanti.

Nome del campo	**Tipo di dato**	**Descrizione**
Portfolio	String	Identificativo del portfolio di appartenenza
DealNumber	Int	Identificativo univoco del trade considerato
AssetType	String	Categoria dell'asset (Option/Stock)
OptionStyle	String	Stile dell'opzione (European/Bermuda)
Underlying	String	Ticker dell'underlying dell'opzione
OptionType	Double	1.0 se si tratta di call e -1.0 se si tratta di put
Currency	String	Valuta del contratto finanziario
Amount	Double	Quantità del contratto finanziario
Volatility	Double	Volatilità dell'underlying in percentuale
Strikes	Array[Double]	Prezzo di esercizio dell'opzione
UnderlyingPrice	Double	Prezzo corrente dell'underlying
Maturity	Double	Tempo alla scadenza dell'opzione in anni
ExerciseDates	Array[Double]	Date di esercizio per opzioni Bermuda
RiskFreeRate	Double	Tasso di interesse

Tabella 4.1: Descrizione dei campi del file sample_trades.csv

Oltre alla lettura del csv, la classe `CsvParser` si occupa del mapping tra i sottostanti e i relativi bucket di rischio. Questa operazione avviene attraverso la lettura dell'apposito file csv `underlying_bucket_mapping.csv`, che associa a ciascun sottostante il bucket di riferimento, insieme ad altre informazioni chiave. Questo file csv, che si trova nelle risorse del progetto, implementa quanto descritto nella tabella 2.1, ovvero associa ogni sottostante al relativo bucket di appartenenza. I campi chiave del file sono contenuti nella tabella 4.2. Il contenuto del csv è ovviamente variabile, per questo esempio sono stati

[1] Con il termine *parsing* si intende il processo di lettura e analisi sintattica di un file o una stringa di testo, con lo scopo di estrarre e interpretare le informazioni in esso contenute. Nel contesto di questa implementazione, il parsing si riferisce alla conversione delle righe del file CSV in oggetti `Trade`, garantendo che ciascun valore venga interpretato correttamente nel formato richiesto (ad esempio, convertendo numeri da stringhe a `double` o `int`).

inseriti 30 *underlyings* in modo da coprire, con un paio di sottostanti, tutte le tipologie di *buckets*.

Nome del campo	**Tipo di dato**	**Descrizione**
Underlying	String	Nome o ticker del sottostante
Bucket	Int	Cateogria di appartenenza (da 1 a 13)
Capitalizzazione	String	Informazione sulla capitalizzazione del sottostante (Grande o Piccola)
Economia	String	Classificazione geografica o economica del sottostante (avanzata o emergente)
Settore	String	Settore industriale o economico di appartenenza del sottostante

Tabella 4.2: Descrizione dei campi del file underlying_bucket_mapping.csv

Infine, la classe `CsvParser` si occupa dell'assegnazione dei fattori di rischio `riskFactorDelta` e `riskFactorVega` per ciascun `trade`, in base alla tipologia dello strumento finanziario.
In particolare:

- Se lo strumento è un'azione (attributo `AssetType` pari ad "Stock"), viene assegnato solo `riskFactorDelta`, che rappresenta il prezzo spot del sottostante ed è identificato con la stringa "Spot"-Underlying.
- Se lo strumento è un'opzione (Attributo `AssetType` pari ad "Option"), vengono assegnati entrambi i fattori di rischio:
  - `riskFactorDelta` segue la stessa logica delle azioni.
  - `riskFactorVega` rappresenta la volatilità implicita dell'opzione e viene calcolato interpolando la volatilità del sottostante sui *tenors* regolamentari. Questo valore è assegnato con la stringa "ImpliedVol"-Underlying-Tenor.

Questa parte di codice garantisce l'associazione di ogni trade al corretto risk factor, ovvero prezzo spot per il delta e curvature, volatilità implicita mappata per il vega.
L'interpolazione della volatilità implicita è effettuata dalla classe `VolatilityInterpolator`, che sarà descritta nel paragrafo successivo.

VolatilityInterpolator

La classe `VolatilityInterpolator` si occupa della mappatura della volatilità implicita delle opzioni su tenors regolamentari. Questo passaggio è essenziale per garantire la conformità alla normativa, che prevede di associare il calcolo del requisito vega a pillars predefiniti.

Per ogni `trade` contenente un'opzione, la classe determina il tenor più vicino alla maturity dell'opzione, scegliendo il valore più alto tra quelli disponibili. I tenors regolamentari, previsti in (3) e qui utilizzati sono:

- 0.5 anni
- 1.0 anno
- 3.0 anni
- 5.0 anni
- 10.0 anni

La selezione del tenor regolamentare più vicino avviene tramite il metodo `getNearestRegulatoryTenor()`, che utilizza una discretizzazione temporale basata sulla classe `TimeDiscretizationFromArray` della libreria finmath. In particolare, il metodo utilizzato per l'assegnazione funziona come di seguito:

$$\text{tenor} = \begin{cases} T^- & \text{se } |maturity - T^-| < |maturity - T^+| \\ T^+ & \text{altrimenti} \end{cases} \tag{4.1}$$

Dove:

- T^- è il tenor regolamentare inferiore, ottenuto con `getTimeIndexNearestLessOrEqual()`.
- T^+ è il tenor regolamentare superiore, ottenuto con `getTimeIndexNearestGreaterOrEqual()`.

Il valore ottenuto viene poi utilizzato per costruire la stringa che rappresenta il fattore di rischio vega, nel formato *"ImpliedVol"-Underlying-Tenor*, ad esempio:

- `ImpliedVol-AAPL-3.0Y`
- `ImpliedVol-TSLA-5.0Y`

Questa operazione è eseguita dal metodo `getFormattedRiskFactorVega()`, che restituisce la stringa correttamente formattata.
Questa stringa viene poi utilizzata all'interno della classe `CsvParser` per assegnare il risk factor corretto a ciascun `trade`, garantendo che il calcolo delle sensibilità vega sia conforme agli standard richiesti.

Trade

La classe `Trade` rappresenta un'operazione finanziaria all'interno del portafoglio, raccogliendo tutte le informazioni necessarie per la sua identificazione, valutazione e gestione del rischio. Ogni istanza della classe corrisponde a uno strumento finanziario, che può essere appunto un'opzione europea, un'opzione bermuda o un'azione, e contiene le caratteristiche contrattuali e i parametri dei modelli di pricing utilizzati.

Oltre ai parametri iniziali, la classe integra dinamicamente i risultati derivanti dai calcoli di pricing e di rischio. Il valore teorico dello strumento è memorizzato nell'attributo `Value`, mentre le sensibilità `DeltaAAD` e `VegaAAD` vengono calcolate dall'apposita classe `AADPricer`. Per la valutazione del rischio di curvatura, sono presenti anche i parametri `CurvatureRiskPlus` e `CurvatureRiskMinus`, calcolati anch'essi dalla relativa classe `CurvatureRiskCalculator`.

La classe offre inoltre metodi per aggiornare alcuni parametri chiave durante il processo di calcolo del rischio. In particolare, il metodo `setCurvatureRisk()` consente di impostare i valori del Curvature Risk dopo la loro elaborazione. Tutti gli attributi della classe sono accessibili tramite metodi *getter*, mentre il metodo `toString()` consente di ottenere una rappresentazione leggibile dell'oggetto, utile per il debugging e il monitoraggio dei dati di input e output.

La tabella 4.3 riassume gli attributi principali della classe `Trade`.

AADPricer

La classe `AADPricer` è progettata per il calcolo delle sensibilità delle opzioni utilizzando l'AAD, sfruttando la libreria finmath per implementare una simulazione Monte Carlo basata sul modello di *Black-Scholes*. Il metodo chiave della classe è rappresentato da `priceAndCalculateGreeks()`, in cui avviene sia il calcolo di delta e vega utilizzando adjoint automatic differentiation che con differenze finite.

L'inizializzazione della simulazione Monte Carlo avviene attraverso il modello di Black-Scholes, che richiede tre parametri fondamentali: il prezzo iniziale del sottostante, il tasso di interesse privo di rischio e la volatilità. Per consentire l'uso di AAD, questi parametri vengono modellati come `RandomVariableDifferentiableAAD`, utilizzando appunto la classe apposita, già implementata in finmath. Il prezzo del sottostante evolve secondo il moto Browniano standard, generato mediante il metodo di *Mersenne Twister*, e la sua traiettoria è simulata utilizzando la discretizzazione di Eulero. L'equazione differenziale stocastica di Black-Scholes che governa il processo del sottostante è riportata di seguito per completezza.

$$dS_t = rS_t dt + \sigma S_t dW_t \tag{4.2}$$

Nome dell'attributo	Tipo di dato
portfolio	String
dealNumber	Int
assetType	String
optionStyle	String
riskFactorDelta	String
riskFactorVega	String
underlying	String
bucket	Int
optionType	Double
currency	String
amount	Double
volatility	Double
strikes	Array[Double]
underlyingPrice	Double
maturity	Double
exerciseDates	Array[Double]
riskFreeRate	Double
value	Double
delta	Double
vega	Double
curvatureRiskPlus	Double
curvatureRiskMinus	Double

Tabella 4.3: Attributi della classe Trade

dove S_t è il prezzo del sottostante, r il tasso di interesse privo di rischio, σ la volatilità e W_t un processo di Wiener. Per ottenere una soluzione numerica, viene utilizzata la discretizzazione di Eulero:

$$S_{t+\Delta t} = S_t \left(1 + r\Delta t + \sigma\sqrt{\Delta t}Z\right) \tag{4.3}$$

dove $Z \sim \mathcal{N}(0,1)$è una variabile casuale normale standard.
La valutazione delle opzioni europee segue la formulazione classica del modello di Black-Scholes, che esprime il valore dell'opzione come l'aspettativa scontata del payoff sotto la misura di rischio neutrale:

$$U(T) := \max(S_T - K, 0) \tag{4.4}$$

$$U_0 = \mathbb{E}^{\mathbb{Q}}\left[e^{-rT}U(T) \mid \mathcal{F}_0\right] \tag{4.5}$$

dove S_T è il prezzo del sottostante alla scadenza, K il prezzo di esercizio e r il tasso di interesse.
Per le opzioni Bermuda, la valutazione segue un approccio backward,riprendendo quanto presente nel lavoro di Fries (10):

$$U_{n+1} := 0 \tag{4.6}$$

$$U_i := B_i \left(\mathbb{E}^{\mathbb{Q}} \left(U_{i+1} | \mathcal{F}_{T_i} \right), U_{i+1}, V_i \right) \tag{4.7}$$

dove B_i è una funzione arbitraria che definisce il criterio di esercizio ottimale. Il criterio di esercizio per le opzioni Bermuda è descritto dalla funzione:

$$G(y, u, v) := \begin{cases} u, & \text{se } y > 0 \\ v, & \text{altrimenti} \end{cases} \tag{4.8}$$

Il valore atteso condizionato della continuazione viene stimato mediante regressione Monte Carlo, utilizzando il metodo di *Longstaff-Schwartz*, già implementato nella classe `BermudanOption` della libreria finmath. L'uso di questa approssimazione è selezionato specificando il parametro `BermudanOption.ExerciseMethod.ESTIMATE\_COND\_EXPECTATION` nel costruttore della classe. Dove i parametri assegnati di default sono: `5`, che rappresenta il numero di funzioni base utilizzate nella regressione Monte Carlo per stimare il valore di continuazione; `false`, il che significa che la base polinomiale non include il payoff max(S-K,0), ma utilizza direttamente il prezzo dell'underlying (S) per costruire la regressione; e infine `false`, indicando che le funzioni base sono polinomi di grado crescente (1, S, S^2, ...) anziché un'approssimazione a intervalli (binning).
Una volta ottenuto il valore degli strumenti, è possibile effettuare il calcolo delle derivate con poche righe di codice. I gradienti delle variabili differenziabili vengono estratti attraverso:

```
1 Map<Long, RandomVariable> derivative = ((
    RandomVariableDifferentiableAAD) value).getGradient();
2 deltaAAD = (derivative.get(initialValue.getID()).getAverage()
    ) * trade.getUnderlyingPrice();
3 vegaAAD = (derivative.get(volatility.getID()).getAverage()) *
     trade.getVolatility();
```

Per le opzioni europee, `AADPricer` include un confronto con le formule analitiche di B&S per il calcolo delle greche, nel caso di un'opzione call si ha:

$$\text{Delta} = N(d_1) \tag{4.9}$$

$$\text{Vega} = S_0 \sqrt{T} \mathcal{N}'(d_1) \tag{4.10}$$

dove:

$$d_1 = \frac{\ln(S_0/K) + (r + \frac{1}{2}\sigma^2)T}{\sigma\sqrt{T}} \tag{4.11}$$

Queste formule offrono un riferimento teorico per validare i risultati ottenuti con la simulazione Monte Carlo e i calcoli di sensibilità basati su AAD. Una volta calcolati i valori di `value`, `deltaAAD` e `vegaAAD`, viene creata una nuova istanza di `Trade` contenente questi valori aggiornati. In questo modo l'oggetto `Trade` include tutte le informazioni necessarie per il calcolo del curvature risk e per le successive aggregazioni; può essere dunque inserito nella lista `updatedTrades` per essere restituito alla fine del metodo.

CurvatureRiskCalculator

La classe `CurvatureRiskCalculator` è progettata per determinare il Curvature Risk, che rappresenta la componente di rischio legata alle variazioni non lineari del valore degli strumenti finanziari con opzionalità (in questo caso opzioni europee e bermuda), non completamente catturate dal delta. Per i dettagli relativi a questo rischio si rimanda alla sezione corrispondente nel Capitolo 1 (2.3.1). Tale formula necessita del contributo del delta calcolato precedentemente. Il calcolo avviene applicando scenari di shock ai prezzi degli underlying e confrontando il valore dell'opzione prima e dopo tali variazioni. La metodologia di pricing utilizzata è la stessa impiegata nella classe `AADPricer`, ma in questo contesto non è necessario l'uso di variabili `RandomVariableDifferentiableAAD`, poiché il rischio di curvatura non viene calcolato tramite AAD.
Infatti, a differenza del calcolo del delta e del vega, che richiede la determinazione delle sensitività tramite differenziazione automatica all'interno del grafo computazionale, il rischio di curvatura è ottenuto confrontando direttamente i valori delle opzioni in scenari di shock predefiniti. Poiché l'approccio non si basa sulla propagazione delle derivate, l'utilizzo di `RandomVariableDifferentiableAAD` sarebbe superfluo. Invece, vengono utilizzate variabili `RandomVariableFromDoubleArray`, che permettono di gestire i valori del modello senza necessità di costruire un grafo computazionale per il calcolo delle derivate, impegnando ulteriore memoria.
Il costruttore della classe, analogamente a quello di `AADPricer`, richiede quattro parametri essenziali per la simulazione: il numero di percorsi Monte Carlo, il numero di passi temporali, il valore del passo temporale e un seed per la generazione casuale. L'obiettivo principale è mantenere la coerenza con la metodologia di pricing adottata per il calcolo del delta e del vega, assicurando al contempo l'indipendenza della valutazione del rischio di curvatura attraverso una classe dedicata.
Il metodo principale `calculateCurvatureRisk()` gestisce l'intero processo di determinazione del rischio di curvatura. I trade vengono inizialmente raggruppati per risk factor (lo stesso utilizzato per il delta risk), selezionando solo quelli con campo `AssetType` pari alla stringa *"Option"*, poiché il Curvature Risk è rilevante esclusivamente per strumenti con

opzionalità. Per ciascun sottostante individuato, viene determinato il bucket regolamentare associato e il relativo peso di rischio, ottenuto tramite il metodo `getCurvatureRiskWeight()`, che interroga la classe `SensitivityAggregator`. Questi parametri sono essenziali per stabilire l'entità dello shock da applicare ai prezzi del sottostante, il cui valore è stabilito dalla normativa.
Definiti i parametri regolatori, vengono applicati due scenari di shock ai prezzi degli underlying: uno scenario positivo, in cui il valore del sottostante aumenta relativamente di una percentuale pari al risk weight associato al relativo bucket per l'aggregazione del delta, e uno scenario negativo, in cui il valore si riduce della stessa quantità; per i valori di shock si può fare riferimento alla tabella 2.2. Per ogni trade, l'opzione viene valutata in tre condizioni: nello scenario base (senza shock), nello scenario con prezzo dell'underlying aumentato e in quello con prezzo diminuito. Il pricing è delegato al metodo `priceOption()`, che implementa la valutazione attraverso simulazioni Monte Carlo, riprendendo la logica già descritta nella classe `AADPricer` per le opzioni europee ed americane.
Dopo aver ottenuto i valori dell'opzione nei tre scenari di prezzo, il metodo principale `calculateCurvatureRisk()` determina il contributo di ciascun trade al Curvature Risk. I valori risultanti vengono aggiornati direttamente nell'oggetto `Trade`, garantendo che i risultati siano disponibili per le successive fasi di aggregazione e gestione del rischio. Infine, i valori aggregati per ciascun sottostante vengono memorizzati nella mappa `curvatureRisk`, che associa ogni sottostante ai rispettivi valori per il bucket di rischio. Entrambi i risultati, relativi allo shock positivo e negativo, vengono conservati poiché verranno utilizzati nella fase successiva di aggregazione, garantendo un'analisi completa del rischio di curvatura.

SensitivityAggregator

La classe `SensitivityAggregator` gestisce il calcolo e l'aggregazione delle componenti di rischio regolamentari appena calcolate. Il suo obiettivo è quello di calcolare le sensibilità nette e ponderate, applicare le correlazioni intra-bucket e inter-bucket e determinare il capitale regolamentare richiesto. Un ulteriore aspetto fondamentale è la gestione della correlazione tra sensibilità, che varia a seconda dello scenario regolamentare selezionato (*medium*, *high*, *low*). Per garantire un'elaborazione efficiente e strutturata dei dati, la classe utilizza `Java Streams`, questi infatti permettono di gestire i diversi livelli delle mappe richiesti dalla normativa per effettuare i raggruppamenti previsti. L'uso di `Collectors.groupingBy()` e `Collectors.summingDouble()` consente di aggregare rapidamente le sensibilità per bucket e fattore di rischio, migliorando la leggibilità e la scalabilità del codice rispetto a un approccio iterativo tradizionale.
L'architettura della classe è supportata dalla mappa `RISK_WEIGHTS`, che associa a ciascun bucket i pesi di rischio regolamentari per delta e vega, si ricorda infatti che i pesi di rischio associati al curvature risk sono i medesimi del delta. Questa mappa è utilizzata nei metodi `calculateWeightedSensitivitiesDelta()` e `calculateWeightedSensitivitiesVega()`

per applicare la corretta ponderazione alle sensibilità nette calcolate precedentemente nei metodi `calculateNetSensitivitiesDelta()` e `calculateNetSensitivitiesVega()`. Questi ultimi aggregano i valori di sensibilità per fattore di rischio (che, come spiegato, contiene il riferimento all'underlying) e bucket, ottenendo una struttura dati gerarchica che consente un rapido accesso ai dati.

L'aggregazione intra-bucket viene gestita nei metodi `aggregateIntraBucketDelta()` e `aggregateIntraBucketVega()`. La logica seguita è quella di raccogliere tutte le sensibilità ponderate per ciascun bucket e applicare la correlazione interna corretta. Questa correlazione varia in base allo scenario regolamentare scelto e viene recuperata dal metodo `getIntraBucketCorrelation()`, che restituisce il valore della correlazione in funzione del bucket e dello scenario selezionato. Per lo scenario *medium*, viene utilizzato il valore base della correlazione, mentre per lo scenario *high* e *low*, vengono applicati i fattori moltiplicativi definiti nel capitolo 1.

Per quanto riguarda l'aggregazione inter-bucket, il metodo `aggregateInterBucket()` combina i contributi di ciascun bucket applicando la correlazione regolamentare, che viene recuperata dal metodo `getInterBucketCorrelation()`. Anche in questo caso, la correlazione varia in base allo scenario: *medium*, *high* o *low*. Il valore finale del capitale regolamentare viene calcolato combinando i contributi dei singoli bucket e le correlazioni inter-bucket per riflettere la diversificazione del portafoglio.

Un aspetto aggiuntivo gestito dalla classe è l'aggregazione del curvature risk, che viene eseguita nei metodi `aggregateCurvatureRiskIntraBucket()` e `aggregateCurvatureRiskInterBucket()`. L'aggregazione intra-bucket del curvature risk segue una logica simile a quella del delta e del vega, ma utilizza correlazioni più elevate, che vengono modificate elevando al quadrato le correlazioni standard. Anche in questo caso, i coefficienti variano in base allo scenario regolamentare scelto e vengono recuperati dal metodo `getAdjustedCorrelationIntra()`. Durante questa prima aggregazione, si determina quale scenario (*plus* o *minus*) sia più rilevante per ciascun bucket e lo si memorizza nella mappa `selectedScenarioByBucket`, che poi viene utilizzata nell'aggregazione tra buckets diversi. Quest'ultima segue un approccio simile, applicando la correlazione regolamentare, recuperata grazie al metodo `getAdjustedCorrelationInter()`, tra bucket per ottenere il capitale totale del rischio di curvatura.

L'implementazione della classe è stata progettata per garantire efficienza computazionale, utilizzando strutture dati ottimizzate e operazioni di aggregazione eseguite tramite *Java Streams*. La suddivisione dei metodi consente una chiara separazione tra i vari passaggi del calcolo, rendendo il codice modulare e facilmente estendibile. Grazie a questa organizzazione, la classe `SensitivityAggregator` rappresenta un componente essenziale per il calcolo del rischio regolamentare, permettendo di gestire in modo efficace le sensibilità e le correlazioni in base agli scenari regolamentari previsti dalla normativa FRTB.

Main

La classe `Main` rappresenta il punto di ingresso dell'applicazione e gestisce l'intero flusso di calcolo delle sensibilità e del requisito di capitale regolamentare secondo la normativa FRTB. Il suo compito principale è orchestrare la lettura dei dati, il calcolo delle sensibilità e l'aggregazione del rischio, garantendo che tutti i passaggi avvengano correttamente e che i risultati siano coerenti con le regole regolamentari previste.

L'esecuzione inizia con la lettura dei file CSV contenenti le informazioni sui bucket e sui trade. Il metodo `parseUnderlyingBucketCsv()` viene utilizzato per caricare la mappatura tra gli underlying e i relativi bucket regolamentari, mentre `parseCsvTrade()` elabora i dati dei trade. Se i file non sono presenti o risultano vuoti, il programma segnala l'errore e termina l'esecuzione.

Una volta caricati i trade, viene eseguito il calcolo delle sensibilità utilizzando la classe `AADPricer`. Dopo aver istanziato l'oggetto `AADPricer` con i parametri della simulazione Monte Carlo, il metodo `priceAndCalculateGreeks()` viene chiamato per calcolare le sensibilità delta e vega per ciascun `trade`. Le sensibilità calcolate vengono quindi associate ai trades corrispondenti e aggiornate con i nuovi valori.

Successivamente, il programma richiama i metodi della classe `SensitivityAggregator` per calcolare e aggregare le sensibilità nette e ponderate per ciascun bucket e fattore di rischio. Il metodo `calculateNetSensitivitiesDelta()` esegue la somma delle sensibilità delta per ogni fattore di rischio e bucket, mentre `calculateNetSensitivitiesVega()` esegue lo stesso calcolo per il vega. Le sensibilità ponderate vengono poi determinate applicando i pesi di rischio regolamentari tramite `calculateWeightedSensitivitiesDelta()` e `calculateWeightedSensitivitiesVega()`.

Dopo aver ottenuto le sensibilità ponderate, il programma procede con il calcolo del curvature risk mediante la classe `CurvatureRiskCalculator`. Quest'ultima applica scenari di shock ai prezzi degli underlying e determina il valore dell'opzione in tre condizioni: scenario base, shock positivo e shock negativo. Il metodo `calculateCurvatureRisk()` calcola il contributo di ciascun trade al rischio di curvatura e aggiorna gli oggetti `Trade` con i nuovi valori di `CurvatureRiskPlus` e `CurvatureRiskMinus`.

Con i valori di delta, vega e curvature calcolati, il programma esegue l'aggregazione intra-bucket per ciascuna componente di rischio nei tre scenari regolamentari previsti (*medium*, *high* e *low*). L'aggregazione avviene separatamente per ciascun bucket, considerando la correlazione tra le sensibilità appartenenti allo stesso gruppo di rischio, utilizzando i metodi `aggregateIntraBucketDelta()`, `aggregateIntraBucketVega()` e `aggregateCurvatureRiskIntraBucket()`. Successivamente, viene selezionato il requisito maggiore tra i tre scenari, che costituirà l'input dell'aggregazione inter-bucket.

Quest'ultima rappresenta lo step finale per il calcolo dei requisiti di capitale, durante la quale i risultati dei bucket vengono combinati applicando i coefficienti di correlazione tra bucket distinti in funzione dello scenario considerato. Questa aggregazione viene gestita

attraverso i metodi `aggregateInterBucketDelta()`, `aggregateInterBucketVega()` e `aggregateCurvatureRiskInterBucket()`.

Infine, i risultati finali vengono stampati a schermo, mostrando il requisito di capitale finale per ciascuna componente di rischio, pari ancora una volta al massimo valore inter-bucket, tra i risultati ottenuti negli scenari *medium*, *high* e *low*.

Di seguito si riporta il diagramma UML del progetto:

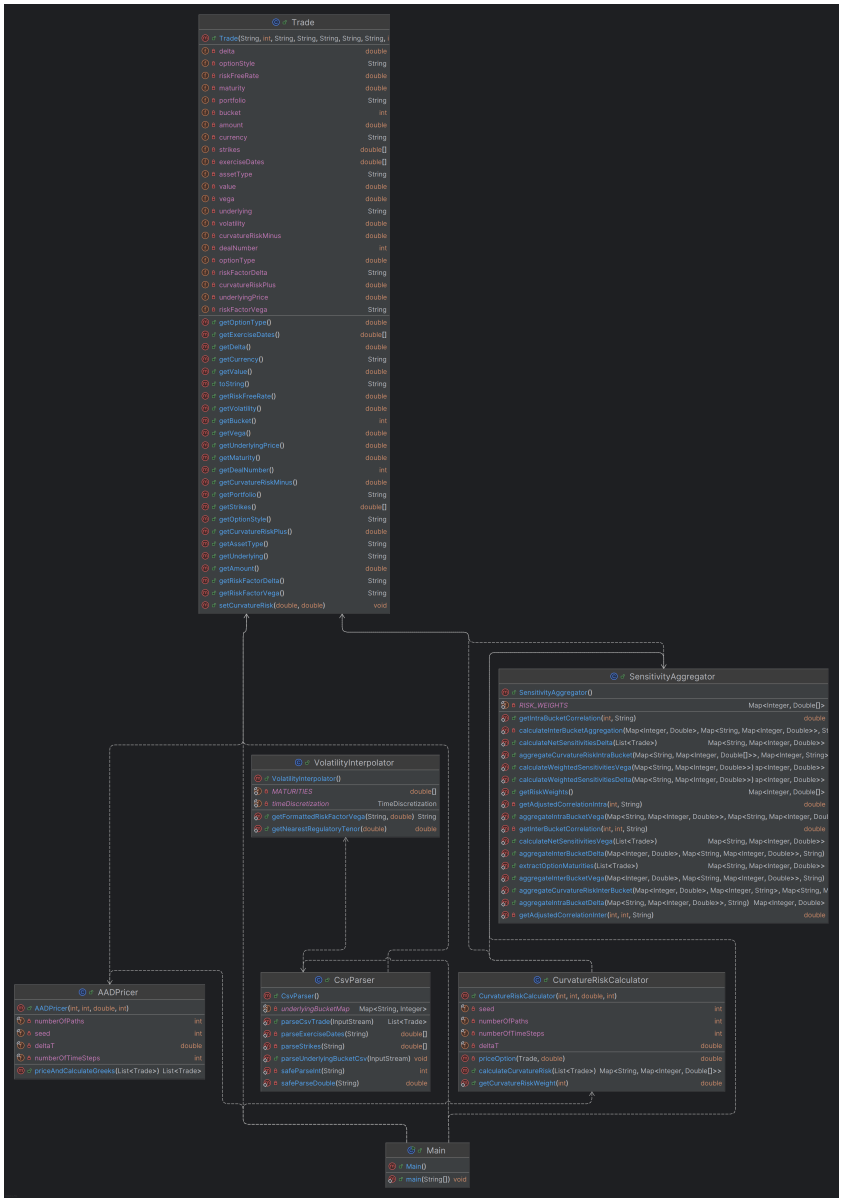

Figura 4.1: Diagramma UML del progetto

4.3 Un esempio di utilizzo

In questa sezione si riporta un esempio di utilizzo del progetto Java finora descritto. Il portafoglio utilizzato per questo test è formato da un totale di 6 trades, con strumenti diversi al fine di coprire le 3 tipologie supportate dall'implementazione. Le caratteristiche di questi sono state assegnate a titolo esemplificativo e, pertanto, non rispecchiano quelle di strumenti equivalenti quotati ad oggi sul mercato. La logica dietro la costruzione di tale csv di ingresso è coprire le tre tipologie di strumenti ed assegnare gli underlying in modo da fornire un esempio di aggregazione intra-bucket e inter-bucket; per questo motivo, sono stati selezionati 6 sottostanti differenti ma appartenenti a sole due tipologie di bucket. In questo modo si evidenzia il processo di aggregazione successivo al pricing e al calcolo delle metriche di rischio. Allo stesso modo per rendere più semplice il test, viene considerato un `amount` pari a 1 contratto di tipo *long*, mentre il valore degli strumenti è assunto essere in USD, ovvero l'ipotetica valuta di reporting della banca fittizia.
Il contenuto del portafoglio è strutturato come segue:

(1) **Azioni:**
   - **Walmart (WMT)** con un prezzo di 90 USD.
   - **JPMorgan Chase (JPM)** con un prezzo di 100 USD.

(2) **Opzioni Europee:**
   - **Call su Apple (AAPL)** con:
     - Strike: 100 USD
     - Prezzo sottostante: 100 USD
     - Volatilità implicita: 32%
     - Scadenza: 1.5 anni
     - Tasso risk-free: 2%
   - **Call su Amazon (AMZN)** con:
     - Strike: 110 USD
     - Prezzo sottostante: 90 USD
     - Volatilità implicita: 25%
     - Scadenza: 2.0 anni
     - Tasso risk-free: 2%

(3) **Opzioni Bermuda:**
   - **Call su Microsoft (MSFT)** con:

  - Strike: 100 USD e 120 USD
  - Prezzo sottostante: 100 USD
  - Volatilità implicita: 30%
  - Scadenza: 2.0 anni
  - Date di esercizio: 1.0 e 2.0 anni
  - Tasso risk-free: 2%
- **Call su Netflix (NFLX)** con:
  - Strike: 100 USD e 90 USD
  - Prezzo sottostante: 100 USD
  - Volatilità implicita: 32%
  - Scadenza: 2.0 anni
  - Date di esercizio: 1.5 e 2.0 anni
  - Tasso risk-free: 2%

Come anticipato la classe `Main` rappresenta il core del progetto in quanto, in essa, vengono eseguite nell'ordine le varie fasi, richiamando le classi necessarie. Il primo passo, eseguito grazie a `CSVParser`, è ovviamente la lettura del file csv `sample_trades.csv` contenente gli strumenti appena elencati, in modo da creare le istanze di `Trade`. In questa fase avvengono altri due passaggi fondamentali per rispettare quanto previsto dalla normativa. Il primo di questi si traduce nella lettura del file `underlying_bucket_mapping.csv` e nella conseguente assegnazione del corretto bucket ad ogni posizione. Inoltre, ogni trade viene caratterizzato dai relativi risk factors, che possono essere solo di tipo "SPOT", come avviene per le azioni, o anche "ImpliedVol" se si tratta di opzioni. Per quest'ultima avviene anche la mappatura dei tenors regolamentari, come descritto nella sezione relativa alla classe `VolatilityInterpolator`. Una volta terminate queste associazioni, i trades vengono stampati a schermo per una verifica delle informazioni.
La fase successiva è relativa al pricing e al calcolo del delta e vega. Prima di richiamare la classe `AADPricer`, predisposta appunto allo svolgimento di tali compiti, vengono definiti i parametri della simulazione utilizzata per il pricing, che in questo esempio sono le seguenti:

- numberOfPaths pari a 100000
- numberOfTimeSteps pari a 100
- deltaT pari a 0.1

Tali parametri sono quelli presenti all'interno del costruttore di AADPricer, che viene chiamato successivamente. L'output relativo a questa fase è riportato di seguito:

```
=== CALCOLO DELLE SENSITIVITIES ===
Calcolando trade: 1
Underlying WMT | Bucket: 5 | AssetType: Stock | OptionStyle: Stock
Value AAD: 90,000000 | Value FD: 0,000000 | Value Analytic: 0,000000
Delta AAD: 90,000000 | Delta FD: 0,000000 | Analytic Delta: 0,000000
Vega AAD: 0,000000 | Vega FD: 0,000000 | Analytic Vega: 0,000000
Time AAD: 0,000 ms | Time FD: 0,000 ms | Time Analytic: 0,000 ms

Calcolando trade: 2
Underlying JPM | Bucket: 8 | AssetType: Stock | OptionStyle: Stock
Value AAD: 100,000000 | Value FD: 0,000000 | Value Analytic: 0,000000
Delta AAD: 100,000000 | Delta FD: 0,000000 | Analytic Delta: 0,000000
Vega AAD: 0,000000 | Vega FD: 0,000000 | Analytic Vega: 0,000000
Time AAD: 0,000 ms | Time FD: 0,000 ms | Time Analytic: 0,000 ms

Calcolando trade: 3
Underlying AAPL | Bucket: 8 | AssetType: Option | OptionStyle: European
Value AAD: 16,975404 | Value FD: 16,975404 | Value Analytic: 16,828181
Delta AAD: 60,985109 | Delta FD: 61,460908 | Analytic Delta: 60,738341
Vega AAD: 15,226626 | Vega FD: 15,226528 | Analytic Vega: 15,065393
Time AAD: 955,212 ms | Time FD: 904,902 ms | Time Analytic: 0,000 ms

Calcolando trade: 4
Underlying AMZN | Bucket: 5 | AssetType: Option | OptionStyle: European
Value AAD: 7,301334 | Value FD: 7,301334 | Value Analytic: 7,261207
Delta AAD: 35,392896 | Delta FD: 35,857758 | Analytic Delta: 35,157002
Vega AAD: 12,281135 | Vega FD: 12,281502 | Analytic Vega: 12,214215
Time AAD: 594,264 ms | Time FD: 897,606 ms | Time Analytic: 0,000 ms

Calcolando trade: 5
Underlying MSFT | Bucket: 8 | AssetType: Option | OptionStyle: Bermudan
Value AAD: 14,231052 | Value FD: 14,231052 | Value Analytic: 0,000000
Delta AAD: 59,138392 | Delta FD: 59,896742 | Analytic Delta: 0,000000
Vega AAD: 14,659538 | Vega FD: 14,637574 | Analytic Vega: 0,000000
Time AAD: 1037,958 ms | Time FD: 904,747 ms | Time Analytic: 0,000 ms

Calcolando trade: 6
Underlying NFLX | Bucket: 5 | AssetType: Option | OptionStyle: Bermudan
Value AAD: 24,391596 | Value FD: 24,391596 | Value Analytic: 0,000000
Delta AAD: 71,012533 | Delta FD: 71,391562 | Analytic Delta: 0,000000
```

```
Vega AAD: 15,626644 | Vega FD: 15,622500 | Analytic Vega: 0,000000
Time AAD: 731,907 ms | Time FD: 838,907 ms | Time Analytic: 0,000 ms
```

Come si può notare, nell'output riportato sono presenti le informazioni chiave di ciascun strumento. Per ognuno vengono stampati: il valore dello strumento (calcolato sia con AAD che con FD, anche se, utilizzando una simulazione gemella, non emergono differenze di pricing), i valori di delta e vega ottenuti con entrambi i metodi e, per le opzioni europee, anche il relativo calcolo analitico. Per le opzioni bermuda non è disponibile un confronto con la formula analitica, dato che il loro pricing non ammette una soluzione chiusa. Inoltre, viene riportato il tempo di esecuzione, utile per confrontare l'efficienza computazionale delle diverse tecniche. Ripetendo la simulazione con diversi parametri, si può notare come i valori ottenuti con AAD risultino generalmente migliori rispetto a quelli derivati dalle differenze finite, in linea con la teoria, pur mostrando errori tipici della simulazione Monte Carlo. Anche i tempi di calcolo confermano una maggiore efficienza complessiva di AAD o, quantomeno, tempistiche analoghe a quelle del metodo FD. Questa efficienza, tuttavia, non emerge pienamente in questo utilizzo, poiché il numero di input (prezzo iniziale e volatilità) è limitato. Il vantaggio computazionale di AAD diventa più evidente all'aumentare degli input, quando il numero di parametri su cui calcolare le *sensitivities* è più elevato.
Il passaggio successivo riguarda l'aggregazione dei valori appena calcolati. In questo caso si ottengono i risk-factors contenuti in tabella 4.4:

Deal Number	Risk Factor Delta	Risk Factor Vega
1	Spot-WMT	null
2	Spot-JPM	null
3	Spot-AAPL	ImpliedVol-AAPL-1.0Y
4	Spot-AMZN	ImpliedVol-AMZN-3.0Y
5	Spot-MSFT	ImpliedVol-MSFT-3.0Y
6	Spot-NFLX	ImpliedVol-NFLX-3.0Y

Tabella 4.4: Risk Factors Delta e Vega per ciascun trade

La prima aggregazione riguarda il calcolo delle *Net Sensitivities* per ogni fattore di rischio, come da formula (2.5). In questo caso, essendo associato un solo trade per ogni fattore di rischio, si ottengono dei valori pari a quelli calcolati precedentemente, ovvero:

```
=== NET SENSITIVITIES DELTA ===
RiskFactor: Spot-JPM | Bucket: 8 | Net Delta: 100,000000
RiskFactor: Spot-AAPL | Bucket: 8 | Net Delta: 60,985109
RiskFactor: Spot-NFLX | Bucket: 5 | Net Delta: 71,012533
RiskFactor: Spot-MSFT | Bucket: 8 | Net Delta: 59,138392
RiskFactor: Spot-AMZN | Bucket: 5 | Net Delta: 35,392896
RiskFactor: Spot-WMT | Bucket: 5 | Net Delta: 90,000000

=== NET SENSITIVITIES VEGA ===
RiskFactor: ImpliedVol-AAPL-1.0Y | Bucket: 8 | Net Vega: 15,226626
RiskFactor: ImpliedVol-MSFT-3.0Y | Bucket: 8 | Net Vega: 14,659538
RiskFactor: ImpliedVol-AMZN-3.0Y | Bucket: 5 | Net Vega: 12,281135
RiskFactor: ImpliedVol-NFLX-3.0Y | Bucket: 5 | Net Vega: 15,626644
```

Il prossimo passaggio riguarda invece il calcolo delle *Weighted Sensitivities*, come descritto in (2.6). In questo caso, i pesi di rischio mappati nell'apposita `HashMap` nella classe `sensitivityAggregator`, sono quelli riportati nelle tabelle 2.2 e 2.3.

```
=== WEIGHTED SENSITIVITIES DELTA ===
RiskFactor: Spot-JPM | Bucket: 8 | Weighted Delta: 50,000000
RiskFactor: Spot-AAPL | Bucket: 8 | Weighted Delta: 30,492555
RiskFactor: Spot-NFLX | Bucket: 5 | Weighted Delta: 21,303760
RiskFactor: Spot-MSFT | Bucket: 8 | Weighted Delta: 29,569196
RiskFactor: Spot-AMZN | Bucket: 5 | Weighted Delta: 10,617869
RiskFactor: Spot-WMT | Bucket: 5 | Weighted Delta: 27,000000

=== WEIGHTED SENSITIVITIES VEGA ===
RiskFactor: ImpliedVol-AAPL-1.0Y | Bucket: 8 | Weighted Vega: 11,843270
RiskFactor: ImpliedVol-MSFT-3.0Y | Bucket: 8 | Weighted Vega: 11,402189
RiskFactor: ImpliedVol-AMZN-3.0Y | Bucket: 5 | Weighted Vega: 9,552267
RiskFactor: ImpliedVol-NFLX-3.0Y | Bucket: 5 | Weighted Vega: 12,154403
```

Terminata questa prima fase relativa al delta e vega, si procede con il calcolo del curvature risk, per ogni risk factor di tipo spot, recuperando il valore di `DeltaAAD` calcolato in precedenza e i corrispondenti pesi di rischio delta associati a ciascun bucket; tramite le formule riportate nell'apposita sezione del capitolo 1 ((2.9) e (2.10)), viene calcolato il relativo rischio positivo e negativo:

```
=== CALCOLO DEL CURVATURE RISK ===
RiskFactor: Spot-AAPL |Bucket: 8 | CVR +: -8,885463 | CVR -: -14,019368
RiskFactor: Spot-NFLX |Bucket: 5 | CVR +: -2,724541 | CVR -: -4,186483
RiskFactor: Spot-MSFT |Bucket: 8 | CVR +: -9,869637 | CVR -: -15,638343
RiskFactor: Spot-AMZN |Bucket: 5 | CVR +: -4,085770 | CVR -: -4,227744
```

Come si può notare dall'output, i valori risultano tutti negativi, ciò è coerente rispetto al tipo di posizioni detenute in portafoglio (Long su call). La normativa FRTB richiede infatti un accantonamento solo se lo shock introduce rischi aggiuntivi non già inclusi nel requisito delta, in questo caso, i valori intra-bucket risulteranno dunque pari a zero, così come il requisito finale associato a questa componente di rischio. A questo punto, avendo calcolato anche il curvature risk per ogni trade, viene creata una nuova istanza dell'oggetto `Trade`, che include i valori di `DeltaAAD`, `VegaAAD`, `CVR+` e `CVR-`. Successivamente, si procede con l'aggregazione intra-bucket.
In questa fase, per l'aggregazione del delta, viene richiamato il metodo `aggregateIntraBucketDelta()` della classe `SensitivityAggregator`, al quale vengono forniti in input lo scenario considerato e le sensibilità ponderate precedentemente calcolate. Allo stesso modo, per il rischio vega, viene utilizzato il metodo `aggregateIntraBucketVega()` che, oltre ai parametri sopra citati, richiede anche la mappa `optionMaturities`, necessaria per determinare correttamente i coefficienti di correlazione secondo la formula (2.17).
Infine, l'aggregazione del curvature risk avviene secondo la formula (2.11), tramite il metodo `aggregateCurvatureRiskIntraBucket()`.Questo riceve in ingresso i valori di CVR^+ e CVR^- contenuti nella mappa `curvatureRisk`, lo scenario considerato e la mappa `selectedScenarioByBucket`, che conterrà il riferimento al curvature risk selezionato (positivo o negativo), in modo da utilizzarlo anche nella successiva aggregazione tra buckets. L'output di questa fase è il seguente:

```
=== AGGREGAZIONE INTRA-BUCKET ===

--- Delta ---
Bucket: 5 | Delta (M: 39,593061, H: 40,442746, L: 38,724736)
Bucket: 8 | Delta (M: 72,665777, H: 74,326043, L: 70,966680)

--- Vega ---
Bucket: 5 | Vega (M: 16,370731, H: 16,590878, L: 16,147584)
Bucket: 8 | Vega (M: 17,433313, H: 17,672925, L: 17,190362)

--- Curvature ---
Bucket: 5 | Curvature (M: 0,000000, H: 0,000000, L: 0,000000)
Bucket: 8 | Curvature (M: 0,000000, H: 0,000000, L: 0,000000)
```

In seguito a questa fase, per delta, vega e curvature viene selezionato il valore corrispondente allo scenario che ha prodotto il requisito maggiore, tale valore viene salvato nella rispettiva mappa, ad esempio `capitalByBucketDeltaFinal`. Questo rappresenta infatti l'input per applicare le formule (2.8) e (2.14):

```
=== AGGREGAZIONE INTER-BUCKET ===

--- Delta ---
(M: 90,181645, H: 91,520037, L: 88,823088)

--- Vega ---
(M: 25,754159, H: 26,118930, L: 25,384147)

--- Curvature ---
(M: 0,000000, H: 0,000000, L: 0,000000)
```

Infine, vengono selezionati i requisiti maggiori tra quelli ottenuti a seguito dell'aggregazione inter-bucket. Questi rappresentano i tre distinti requisiti di capitale:

```
=== RISULTATI FINALI ===
Final Capital Requirement Delta: 91,520037
Final Capital Requirement Vega: 26,118930
Final Capital Requirement Curvature: 0,000000

=== FINE DEL PROGRAMMA ===
```

5 Conclusioni

Questo progetto ha esplorato un'alternativa al metodo delle differenze finite per il calcolo delle sensitivities richieste dalla normativa FRTB. L'obiettivo non era certamente modificare un quadro regolamentare ormai consolidato e in via di applicazione, ma quanto dimostrare come una tecnica più precisa ed efficiente, quale l'AAD, possa fornire risultati affidabili e replicabili. L'integrazione di questa metodologia nei sistemi di pricing già in uso potrebbe migliorare il processo di calcolo delle sensitivities, ottimizzando la gestione del rischio senza alterare le procedure operative esistenti.
L'implementazione proposta presenta alcune limitazioni, legate principalmente all'adattamento delle metodologie di calcolo richieste dal regolatore e alla selezione dei prodotti finanziari analizzati. Il modello sviluppato, infatti, si è concentrato esclusivamente su un portafoglio semplificato, composto da opzioni europee, opzioni bermuda e azioni, escludendo strumenti più complessi. Inoltre, le ipotesi adottate nella costruzione del framework non coprono certamente tutte le casistiche inerenti le classi di rischio, rendendo necessaria un'estensione futura del modello per aumentarne l'applicabilità e l'affidabilità. Tuttavia, queste semplificazioni non ne pregiudicano la validità concettuale, ma piuttosto pongono le basi per futuri sviluppi. Un esempio potrebbe riguardare l'estensione del metodo a un insieme più ampio di strumenti finanziari e classi di rischio, prima tra tutte la gestione del rischio valutario di posizioni denominate in una valuta differente da quella di reporting della banca.
In conclusione, l'analisi svolta suggerisce che l'AAD possa offrire un approccio alternativo per il calcolo delle sensitivities, con potenziali benefici in termini di efficienza e precisione. Sebbene siano necessari ulteriori approfondimenti e validazioni, i risultati ottenuti mostrano spunti interessanti che potrebbero essere esplorati in ricerche future nel campo della gestione del rischio finanziario. Oltre a rappresentare un'importante opportunità di approfondimento del linguaggio Java, questo lavoro è stato per me un'occasione per comprendere più a fondo la normativa e valutare come questa possa essere applicata in modo talvolta più efficiente, non limitandosi alla mera esecuzione delle norme emesse dal regolatore.

Bibliografia

[1] Antonov, A. (2017). Algorithmic differentiation for callable exotics. *Available at SSRN 2839362*.

[2] Basel Committee on Banking Supervision (2017). Simplified alternative to the standardised approach to market risk capital requirements. Available at https://www.bis.org.

[3] Basel Committee on Banking Supervision (2019). Minimum capital requirements for market risk. Available at https://www.bis.org.

[4] Basel Committee on Banking Supervision (2022). Explanatory note on the minimum capital requirements for market risk. Available at https://www.bis.org.

[5] Capriotti, L. and Giles, M. B. (2011). Algorithmic differentiation: Adjoint greeks made easy. *Available at SSRN 1801522*.

[6] European Banking Authority (2017). Discussion paper on eu implementation of mkr and ccr revised standards. Available at https://www.eba.europa.eu.

[7] European Parliament and Council (2024). Regulation (eu) 2024/1623 of the european parliament and of the council of 31 may 2024 amending regulation (eu) no 575/2013 as regards requirements for credit risk, credit valuation adjustment risk, operational risk, market risk and the output floor. *Official Journal of the European Union*. Available at https://eur-lex.europa.eu/eli/reg/2024/1623/oj/eng.

[8] Fries, C. P. (2017). Automatic backward differentiation for american monte-carlo algorithms (conditional expectation). *arXiv preprint arXiv:1707.04942*.

[9] Fries, C. P. (2019). Stochastic automatic differentiation: automatic differentiation for monte-carlo simulations. *Quantitative Finance*, 19(6):1043–1059.

[10] Fries, C. P. (2022). Stochastic algorithmic differentiation of (expectations of) discontinuous functions (indicator functions). *International Journal of Computer Mathematics*, 99(2):204–226.

[11] Giering, R. and Kaminski, T. (1998). Recipes for adjoint code construction. *ACM Transactions on Mathematical Software (TOMS)*, 24(4):437–474.

[12] Henrard, M. (2017). *Algorithmic differentiation in finance explained*. Springer.

[13] Longstaff, F. A. and Schwartz, E. S. (2001). Valuing american options by simulation: a simple least-squares approach. *The review of financial studies*, 14(1):113–147.

[14] van den Berg, B., Schrijvers, T., McKinna, J., and Vandenbroucke, A. (2024). Forward- or reverse-mode automatic differentiation: What's the difference? *Science of Computer Programming*, 231:103010.

[15] Wengert, R. E. (1964). A simple automatic derivative evaluation program. *Communications of the ACM*, 7(8):463–464.

Appendice

In questa sezione è riportato il codice utilizzato nel progetto, diviso per classi.

CsvParser

```
1  package it.tesi;
2
3  import org.apache.commons.csv.*;
4  import java.io.*;
5  import java.nio.charset.StandardCharsets;
6  import java.util.ArrayList;
7  import java.util.Arrays;
8  import java.util.HashMap;
9  import java.util.List;
10 import java.util.Map;
11
12 public class CsvParser {
13
14   //Creazione mappa bucket
15   private static final Map<String, Integer>
        underlyingBucketMap = new HashMap<>();
16
17   //Lettura csv per assegnazione buckets
18   public static void parseUnderlyingBucketCsv(InputStream
        inputStream) {
19       try (Reader reader = new BufferedReader(new
            InputStreamReader(inputStream, StandardCharsets.
            UTF_8));
20              CSVParser csvParser = new CSVParser(reader,
                    CSVFormat.DEFAULT
21                          .builder()
22                          .setHeader()
23                          .setIgnoreHeaderCase(true)
24                          .setTrim(true)
25                          .build())) {
26
```

```
27            for (CSVRecord record : csvParser) {
28                String underlying = record.get("Underlying").
                      trim();
29                int bucket = safeParseInt(record.get("Bucket"))
                      ;
30                underlyingBucketMap.put(underlying, bucket);
31            }
32
33        } catch (IOException e) {
34            System.err.println("Errore nella lettura del file
                  CSV dei buckets: " + e.getMessage());
35        }
36    }
37
38    //Lettura csv trades e creazione istanza dell'oggetto Trade
39    public static List<Trade> parseCsvTrade(InputStream
          inputStream) {
40        List<Trade> trades = new ArrayList<>();
41
42        try (Reader reader = new BufferedReader(new
              InputStreamReader(inputStream, StandardCharsets.
              UTF_8));
43               CSVParser csvParser = new CSVParser(reader,
                     CSVFormat.DEFAULT.builder()
44                         .setHeader()
45                         .setIgnoreHeaderCase(true)
46                         .setTrim(true)
47                         .build())) {
48
49            for (CSVRecord record : csvParser) {
50
51                String underlying = record.get("Underlying").
                      trim();
52                int bucket = underlyingBucketMap.getOrDefault(
                      underlying, 0); // Se non trovato, assegna 0
53                  double maturity = safeParseDouble(record.get(
                        "Maturity"));
54
55                  // Assegnazione di riskFactorDelta e
                        riskFactorVega
56                  String riskFactorDelta = "Spot-" + underlying
                        ;
57                  String riskFactorVega = null;
```

```
58
59                  if (record.get("AssetType").equalsIgnoreCase(
                        "Option")) {
60                      riskFactorVega = VolatilityInterpolator.
                            getFormattedRiskFactorVega(underlying,
                             maturity);
61                  }
62
63                Trade trade = new Trade(
64                        record.get("Portfolio"),
65                        safeParseInt(record.get("DealNumber")),
66                        record.get("AssetType"),
67                        record.get("AssetType").
                            equalsIgnoreCase("Stock") ? "Stock"
                            : record.get("OptionStyle"),
68                        riskFactorDelta,
69                        riskFactorVega,
70                        underlying,
71                        bucket,
72                        safeParseDouble(record.get("OptionType"
                            )),
73                        record.get("Currency"),
74                        safeParseDouble(record.get("Amount")),
75                        safeParseDouble(record.get("Volatility"
                            )),
76                        parseStrikes(record.get("Strikes")),
77                        safeParseDouble(record.get("
                            UnderlyingPrice")),
78                        safeParseDouble(record.get("Maturity"))
                            ,
79                        parseExerciseDates(record.get("
                            ExerciseDates")),
80                        safeParseDouble(record.get("
                            RiskFreeRate")),
81                        record.isMapped("Value") && !record.get
                            ("Value").isEmpty() ?
                            safeParseDouble(record.get("Value"))
                             : 0.0,
82                        record.isMapped("Delta") && !record.get
                            ("Delta").isEmpty() ?
                            safeParseDouble(record.get("Delta"))
                             : 0.0,
```

```
83                          record.isMapped("Vega") && !record.get(
                               "Vega").isEmpty() ? safeParseDouble(
                               record.get("Vega")) : 0.0,
84                          record.isMapped("CurvatureRiskPlus") &&
                                !record.get("CurvatureRiskPlus").
                               isEmpty() ? safeParseDouble(record.
                               get("CurvatureRiskPlus")) : 0.0,
85                          record.isMapped("CurvatureRiskMinus")
                               && !record.get("CurvatureRiskMinus")
                               .isEmpty() ? safeParseDouble(record.
                               get("CurvatureRiskMinus")) : 0.0
86                      );
87
88
89                  trades.add(trade);
90              }
91
92          } catch (IOException e) {
93              System.err.println("Errore nella lettura del file
                   CSV: " + e.getMessage());
94          }
95
96          return trades;
97      }
98
99
100     // Metodo helper per gestire conversioni errate di
           Integer
101     private static int safeParseInt(String value) {
102         if (value == null || value.trim().isEmpty()) {
103             return 0;
104         }
105         try {
106             return Integer.parseInt(value.trim());
107         } catch (NumberFormatException e) {
108             return 0;
109         }
110     }
111
112     // Metodo helper per gestire conversioni errate di Double
113     private static double safeParseDouble(String value) {
114         if (value == null || value.trim().isEmpty()) {
115             return 0.0;
```

```
116         }
117         try {
118             return Double.parseDouble(value);
119         } catch (NumberFormatException e) {
120             return 0.0;
121         }
122     }
123 
124     // Metodo per convertire ExerciseDates da String a double
        []
125     private static double[] parseExerciseDates(String dates)
        {
126         if (dates == null || dates.trim().isEmpty()) {
127             return new double[0];
128         }
129         try {
130             return Arrays.stream(dates.split(";"))
131                     .mapToDouble(Double::parseDouble)
132                     .toArray();
133         } catch (NumberFormatException e) {
134             return new double[0];
135         }
136     }
137 
138  // Metodo per convertire gli Strikes da String a double[]
139     private static double[] parseStrikes(String strikes) {
140         if (strikes == null || strikes.trim().isEmpty()) {
141             return new double[0];
142         }
143         try {
144             return Arrays.stream(strikes.split(";"))
145                     .mapToDouble(Double::parseDouble)
146                     .toArray();
147         } catch (NumberFormatException e) {
148             return new double[0];
149         }
150     }
151 
152 }
```

VolatilityInterpolator

```
1  package it.tesi;
2
3  import java.util.Locale;
4
5  import net.finmath.time.TimeDiscretization;
6  import net.finmath.time.TimeDiscretizationFromArray;
7
8  public class VolatilityInterpolator {
9
10     private static final double[] MATURITIES = {0.5, 1.0,
          3.0, 5.0, 10.0};
11     private static final TimeDiscretization
          timeDiscretization = new TimeDiscretizationFromArray(
          MATURITIES);
12
13
14     //Trova il tenor regolamentare piu' vicino per una data
          maturity
15     public static double getNearestRegulatoryTenor(double
          maturity) {
16         int indexGreaterOrEqual = timeDiscretization.
              getTimeIndexNearestGreaterOrEqual(maturity);
17         int indexLessOrEqual = timeDiscretization.
              getTimeIndexNearestLessOrEqual(maturity);
18
19         if (indexGreaterOrEqual < 0) {
20             return MATURITIES[0];
21         } else if (indexGreaterOrEqual >= MATURITIES.length)
              {
22             return MATURITIES[MATURITIES.length - 1];
23         }
24
25         if (indexLessOrEqual >= 0) {
26             double lower = MATURITIES[indexLessOrEqual];
27             double upper = MATURITIES[indexGreaterOrEqual];
28
29             return (Math.abs(maturity - lower) < Math.abs(
                  maturity - upper)) ? lower : upper;
30         }
31
32         return MATURITIES[indexGreaterOrEqual];
```

```
33      }
34
35
36      //Restituisce riskFactorVega formattato, includendo l'
            underlying e il tenor
37      public static String getFormattedRiskFactorVega(String
            underlying, double maturity) {
38          double nearestTenor = getNearestRegulatoryTenor(
                maturity);
39          return String.format(Locale.US, "ImpliedVol-%s-%.1fY"
                , underlying, nearestTenor);
40      }
41  }
```

Trade

```
1  package it.tesi;
2
3  import java.util.Arrays;
4  import java.util.Locale;
5
6  public class Trade {
7      private String portfolio;
8      private int dealNumber;
9      private String assetType;
10     private String optionStyle;
11     private String riskFactorDelta;
12     private String riskFactorVega;
13     private String underlying;
14     private int bucket;
15     private double optionType;
16     private String currency;
17     private double amount;
18     private double volatility;
19     private double[] strikes;
20     private double underlyingPrice;
21     private double maturity;
22     private double[] exerciseDates;
23     private double riskFreeRate;
24     private double value;
25     private double delta;
26     private double vega;
```

```
27      private double curvatureRiskPlus;
28      private double curvatureRiskMinus;
29
30
31      public Trade(String portfolio, int dealNumber, String
            assetType, String optionStyle,
32                   String riskFactorDelta, String
                        riskFactorVega, String underlying, int
                        bucket, double optionType, String
                        currency,
33                   double amount, double volatility, double[]
                        strikes,
34                   double underlyingPrice, double maturity,
                        double[] exerciseDates, double
                        riskFreeRate,
35                   double value, double delta, double vega,
                        double curvatureRiskPlus, double
                        curvatureRiskMinus) {
36          this.portfolio = portfolio;
37          this.dealNumber = dealNumber;
38          this.assetType = assetType;
39          this.optionStyle = optionStyle;
40          this.riskFactorDelta = riskFactorDelta;
41          this.riskFactorVega = riskFactorVega;
42          this.underlying = underlying;
43          this.bucket = bucket;
44          this.optionType = optionType;
45          this.currency = currency;
46          this.amount = amount;
47          this.volatility = volatility;
48          this.strikes = strikes;
49          this.underlyingPrice = underlyingPrice;
50          this.maturity = maturity;
51          this.exerciseDates = exerciseDates;
52          this.riskFreeRate = riskFreeRate;
53          this.value = value;
54          this.delta = delta;
55          this.vega = vega;
56          this.curvatureRiskPlus = curvatureRiskPlus;
57          this.curvatureRiskMinus = curvatureRiskMinus;
58      }
59
60      public String getPortfolio() {
```

```
61          return portfolio;
62      }
63
64      public int getDealNumber() {
65          return dealNumber;
66      }
67
68      public String getAssetType() {
69          return assetType;
70      }
71
72      public String getOptionStyle() {
73          return optionStyle;
74      }
75
76      public String getRiskFactorDelta() {
77          return riskFactorDelta;
78      }
79
80      public String getRiskFactorVega() {
81          return riskFactorVega;
82      }
83
84      public String getUnderlying() {
85          return underlying;
86      }
87
88      public int getBucket() {
89          return bucket;
90      }
91
92      public double getOptionType() {
93          return optionType;
94      }
95
96      public String getCurrency() {
97          return currency;
98      }
99
100     public double getAmount() {
101         return amount;
102     }
103
```

```
104     public double getVolatility() {
105         return volatility;
106     }
107 
108     public double[] getStrikes() {
109         return strikes;
110     }
111 
112     public double getUnderlyingPrice() {
113         return underlyingPrice;
114     }
115 
116     public double getMaturity() {
117         return maturity;
118     }
119 
120     public double[] getExerciseDates() {
121         return exerciseDates;
122     }
123 
124     public double getRiskFreeRate() {
125         return riskFreeRate;
126     }
127 
128     public double getValue() {
129       return value;
130     }
131 
132     public double getDelta() {
133         return delta;
134     }
135 
136     public double getVega() {
137         return vega;
138     }
139 
140   public double getCurvatureRiskPlus() {
141         return curvatureRiskPlus;
142     }
143 
144     public double getCurvatureRiskMinus() {
145         return curvatureRiskMinus;
146     }
```

```
147
148     public void setCurvatureRisk(double plus, double minus) {
149         this.curvatureRiskPlus = plus;
150         this.curvatureRiskMinus = minus;
151     }
152
153     @Override
154     public String toString() {
155         return String.format(Locale.US,
156                 "Trade [Portfolio=%s, DealNumber=%d,
                       AssetType=%s, OptionStyle=%s,
                       RiskFactorDelta=%s, " +
157                 "RiskFactorVega=%s, Underlying=%s, Bucket=%d,
                        OptionType=%.2f, Currency=%s, Amount=%.2f
                       , " +
158                 "Volatility=%.2f, Strikes=%s, UnderlyingPrice
                       =%.2f, Maturity=%.6f, " +
159                 "ExerciseDates=%s, RiskFreeRate=%.3f, Value
                       =%.6f, Delta=%.6f, Vega=%.6f, " +
160                 "CurvatureRiskPlus=%.6f, CurvatureRiskMinus
                       =%.6f]",
161                 portfolio, dealNumber, assetType, optionStyle
                       , riskFactorDelta, riskFactorVega,
                       underlying, bucket,
162                 optionType, currency, amount, volatility,
                       Arrays.toString(strikes), underlyingPrice,
                        maturity,
163                 Arrays.toString(exerciseDates), riskFreeRate,
                        value, delta, vega, curvatureRiskPlus,
                       curvatureRiskMinus);
164     }
165 }
```

AADPricer

```
1 package it.tesi;
2
3 import net.finmath.exception.CalculationException;
4 import net.finmath.functions.AnalyticFormulas;
5 import net.finmath.montecarlo.*;
6 import net.finmath.montecarlo.assetderivativevaluation.*;
```

```
7  import net.finmath.montecarlo.assetderivativevaluation.models
       .BlackScholesModel;
8  import net.finmath.montecarlo.assetderivativevaluation.
       products.BermudanOption;
9  import net.finmath.montecarlo.assetderivativevaluation.
       products.EuropeanOption;
10 import net.finmath.montecarlo.automaticdifferentiation.
       RandomVariableDifferentiable;
11 import net.finmath.montecarlo.automaticdifferentiation.
       backward.RandomVariableDifferentiableAAD;
12 import net.finmath.montecarlo.automaticdifferentiation.
       backward.RandomVariableDifferentiableAADFactory;
13 import net.finmath.montecarlo.process.*;
14 import net.finmath.stochastic.RandomVariable;
15 import net.finmath.time.*;
16
17 import java.util.*;
18
19 public class AADPricer {
20
21     private final int numberOfPaths;
22     private final int numberOfTimeSteps;
23     private final double deltaT;
24     private final int seed;
25
26
27     public AADPricer(int numberOfPaths, int numberOfTimeSteps
           , double deltaT, int seed) {
28         this.numberOfPaths = numberOfPaths;
29         this.numberOfTimeSteps = numberOfTimeSteps;
30         this.deltaT = deltaT;
31         this.seed = seed;
32     }
33
34     public List<Trade> priceAndCalculateGreeks(List<Trade>
           trades) throws CalculationException {
35         List<Trade> updatedTrades = new ArrayList<>();
36         RandomVariableDifferentiableAADFactory
               randomVariableFactory = new
               RandomVariableDifferentiableAADFactory();
37
38         for (Trade trade : trades) {
39             System.out.println("Calcolando trade: " + trade.
```

```
            getDealNumber());
40
41          double deltaAAD = 0.0, deltaFD = 0.0, vegaAAD =
               0.0, vegaFD = 0.0;
42          double analyticValue = 0.0, analyticDelta = 0.0,
               analyticVega = 0.0;
43          double timeAAD = 0.0, timeFD = 0.0, timeAnalytic
               = 0.0;
44
45          RandomVariable value = null;
46          RandomVariable valueOriginal = null;
47          RandomVariable valueUp = null;
48          RandomVariable valueVolUp = null;
49
50          if (trade.getAssetType().equalsIgnoreCase("Stock"
               )) {
51            value = new RandomVariableFromDoubleArray(trade
                 .getUnderlyingPrice());
52             deltaAAD = trade.getUnderlyingPrice();
53             vegaAAD = 0.0;
54
55          }
56          else {
57
58              //Definizione delle variabili differenziabili
59              RandomVariableDifferentiableAAD initialValue
                   = (RandomVariableDifferentiableAAD)
                   randomVariableFactory.createRandomVariable
                   (trade.getUnderlyingPrice());
60              RandomVariableDifferentiable riskFreeRate =
                   randomVariableFactory.createRandomVariable
                   (trade.getRiskFreeRate());
61              RandomVariableDifferentiableAAD volatility =
                   (RandomVariableDifferentiableAAD)
                   randomVariableFactory.createRandomVariable
                   (trade.getVolatility());
62
63              //Creazione del modello Black-Scholes
64              BlackScholesModel model = new
                   BlackScholesModel(initialValue,
                   riskFreeRate, volatility,
                   randomVariableFactory);
65
```

```
66                  //Time discretization e Simulazione MC
67                  TimeDiscretization timeDiscretization = new
                        TimeDiscretizationFromArray(0.0,
                        numberOfTimeSteps, deltaT);
68                  BrownianMotion brownianMotion = new
                        BrownianMotionFromMersenneRandomNumbers(
                        timeDiscretization, 1, numberOfPaths, seed
                        );
69                  EulerSchemeFromProcessModel process = new
                        EulerSchemeFromProcessModel(model,
                        brownianMotion);
70                  MonteCarloAssetModel monteCarloModel = new
                        MonteCarloAssetModel(process);
71
72                  //Calcolo con AAD
73                  long startAAD = System.nanoTime();
74                  if (trade.getOptionStyle().equalsIgnoreCase("
                        European")) {
75                      EuropeanOption europeanOption = new
                            EuropeanOption(trade.getUnderlying(),
                            trade.getMaturity(), trade.getStrikes
                            ()[0], trade.getOptionType());
76                      value = (RandomVariableDifferentiable)
                            europeanOption.getValue(0.0,
                            monteCarloModel);
77                  } else {
78                      double[] exerciseDates = trade.
                            getExerciseDates();
79                      double[] notionals = new double[
                            exerciseDates.length];
80                      double[] strikes = trade.getStrikes();
81
82                      Arrays.fill(notionals, 1.0);
83
84
85                      BermudanOption bermudanOption = new
                            BermudanOption(exerciseDates,
                            notionals, strikes, BermudanOption.
                            ExerciseMethod.
                            ESTIMATE_COND_EXPECTATION);
86                      value = bermudanOption.getValue(0.0,
                            monteCarloModel);
87
```

```
88                  }
89
90                  Map<Long, RandomVariable> derivative = ((
                        RandomVariableDifferentiableAAD) value).
                        getGradient();
91                  deltaAAD = derivative.get(initialValue.getID
                        ()).getAverage() * trade.
                        getUnderlyingPrice();
92                  vegaAAD = (derivative.get(volatility.getID())
                        .getAverage()) * trade.getVolatility();
93                  long endAAD = System.nanoTime();
94                  timeAAD = (endAAD - startAAD) / 1e6;
95
96
97                  // Calcolo con Differenze Finite
98                  long startFD = System.nanoTime();
99
100                 BlackScholesModel modelOriginal = new
                        BlackScholesModel(initialValue,
                        riskFreeRate, volatility,
                        randomVariableFactory);
101                 MonteCarloAssetModel monteCarloOriginal = new
                         MonteCarloAssetModel(new
                        EulerSchemeFromProcessModel(modelOriginal,
                         brownianMotion));
102
103                 if (trade.getOptionStyle().equalsIgnoreCase("
                        European")) {
104                      EuropeanOption europeanOption = new
                            EuropeanOption(trade.getUnderlying(),
                            trade.getMaturity(), trade.getStrikes
                            ()[0], trade.getOptionType());
105                      valueOriginal = (
                            RandomVariableDifferentiable)
                            europeanOption.getValue(0.0,
                            monteCarloOriginal);
106
107                      //Delta
108                      BlackScholesModel modelUp = new
                            BlackScholesModel(initialValue.mult
                            (1.01), riskFreeRate, volatility,
                            randomVariableFactory);
109                      MonteCarloAssetModel monteCarloUp = new
```

```
                    MonteCarloAssetModel(new
                    EulerSchemeFromProcessModel(modelUp,
                    brownianMotion));
110                 valueUp = (RandomVariableDifferentiable)
                    europeanOption.getValue(0.0,
                    monteCarloUp);
111
112                 //Vega
113                 BlackScholesModel modelVolUp = new
                    BlackScholesModel(initialValue,
                    riskFreeRate, volatility.mult(1.01),
                    randomVariableFactory);
114                 MonteCarloAssetModel monteCarloVolUp =
                    new MonteCarloAssetModel(new
                    EulerSchemeFromProcessModel(modelVolUp
                    , brownianMotion));
115                 valueVolUp = (
                    RandomVariableDifferentiable)
                    europeanOption.getValue(0.0,
                    monteCarloVolUp);
116             }
117             else { //Caso Bermudan Option
118                 double[] exerciseDates = trade.
                    getExerciseDates();
119                 double[] notionals = new double[
                    exerciseDates.length];
120                 double[] strikes = trade.getStrikes();
121
122                 Arrays.fill(notionals, 1.0);
123
124                 BermudanOption bermudanOption = new
                    BermudanOption(exerciseDates,
                    notionals, strikes, BermudanOption.
                    ExerciseMethod.
                    ESTIMATE_COND_EXPECTATION);
125
126                 valueOriginal = bermudanOption.getValue
                    (0.0, monteCarloOriginal);
127
128                 //Delta FD
129                 BlackScholesModel modelUp = new
                    BlackScholesModel(initialValue.mult
                    (1.01), riskFreeRate, volatility,
```

```
                randomVariableFactory);
130             MonteCarloAssetModel monteCarloUp = new
                MonteCarloAssetModel(new
                EulerSchemeFromProcessModel(modelUp,
                brownianMotion));
131             valueUp = bermudanOption.getValue(0.0,
                monteCarloUp);
132
133             //Vega FD
134             BlackScholesModel modelVolUp = new
                BlackScholesModel(initialValue,
                riskFreeRate, volatility.mult(1.01),
                randomVariableFactory);
135             MonteCarloAssetModel monteCarloVolUp =
                new MonteCarloAssetModel(new
                EulerSchemeFromProcessModel(modelVolUp
                , brownianMotion));
136             valueVolUp = bermudanOption.getValue(0.0,
                 monteCarloVolUp);
137         }
138
139         // Calcolo del Delta e Vega FD
140         deltaFD = ((valueUp.getAverage() -
                valueOriginal.getAverage()) / 0.01);
141         vegaFD = ((valueVolUp.getAverage() -
                valueOriginal.getAverage()) / 0.01);
142
143         long endFD = System.nanoTime();
144         timeFD = (endFD - startFD) / 1e6;
145
146
147         //Check con formule analitiche
148         if (trade.getOptionStyle().equalsIgnoreCase("
                European")) {
149         boolean isCall = trade.getOptionType() ==
                1.0;
150         analyticValue = AnalyticFormulas.
                blackScholesOptionValue(trade.
                getUnderlyingPrice(), trade.
                getRiskFreeRate(), trade.getVolatility(),
                trade.getMaturity(), trade.getStrikes()
                [0], isCall);
151         double deltaCall = AnalyticFormulas.
```

```
                    blackScholesOptionDelta(trade.
                    getUnderlyingPrice(), trade.
                    getRiskFreeRate(), trade.getVolatility(),
                    trade.getMaturity(), trade.getStrikes()
                    [0]);
152                 analyticDelta = ((trade.getOptionType() ==
                    1.0) ? deltaCall : (deltaCall - 1)) *
                    trade.getUnderlyingPrice();
153                 analyticVega = AnalyticFormulas.
                    blackScholesOptionVega(trade.
                    getUnderlyingPrice(), trade.
                    getRiskFreeRate(), trade.getVolatility(),
                    trade.getMaturity(), trade.getStrikes()
                    [0]) * trade.getVolatility();
154
155                 }
156             }
157
158
159             System.out.printf("Underlying %s | Bucket: %s |
                    AssetType: %s | OptionStyle: %s\n", trade.
                    getUnderlying(), trade.getBucket(), trade.
                    getAssetType(), trade.getOptionStyle());
160             System.out.printf("Value AAD: %.6f | Value FD:
                    %.6f | Value Analytic: %.6f\n", value.
                    getAverage(), (valueOriginal != null ?
                    valueOriginal.getAverage() : 0.0),
                    analyticValue);
161             System.out.printf("Delta AAD: %.6f | Delta FD:
                    %.6f | Analytic Delta: %.6f\n", deltaAAD,
                    deltaFD, analyticDelta);
162             System.out.printf("Vega AAD: %.6f | Vega FD: %.6f
                     | Analytic Vega: %.6f\n", vegaAAD, vegaFD,
                    analyticVega);
163             System.out.printf("Time AAD: %.3f ms | Time FD:
                    %.3f ms | Time Analytic: %.3f ms\n\n", timeAAD
                    , timeFD, timeAnalytic);
164
165             //Aggiorno il Trade con i valori di valueAAD,
                    deltaAAD e vegaAAD
166             Trade updatedTrade = new Trade(
167                     trade.getPortfolio(), trade.getDealNumber
                        (), trade.getAssetType(), trade.
```

```
                        getOptionStyle(),
168                     trade.getRiskFactorDelta(), trade.
                        getRiskFactorVega(), trade.
                        getUnderlying(), trade.getBucket(),
                        trade.getOptionType(),
169                     trade.getCurrency(), trade.getAmount(),
                        trade.getVolatility(),
170                     trade.getStrikes(), trade.
                        getUnderlyingPrice(), trade.
                        getMaturity(), trade.getExerciseDates
                        (),
171                     trade.getRiskFreeRate(), value.getAverage
                        (), deltaAAD, vegaAAD, trade.
                        getCurvatureRiskPlus(), trade.
                        getCurvatureRiskMinus()
172             );
173
174             updatedTrades.add(updatedTrade);
175         }
176
177         return updatedTrades;
178     }
179 }
```

CurvatureRiskCalculator

```
 1 package it.tesi;
 2
 3 import net.finmath.exception.CalculationException;
 4 import net.finmath.montecarlo.*;
 5 import net.finmath.montecarlo.assetderivativevaluation.*;
 6 import net.finmath.montecarlo.assetderivativevaluation.models
     .BlackScholesModel;
 7 import net.finmath.montecarlo.assetderivativevaluation.
     products.BermudanOption;
 8 import net.finmath.montecarlo.assetderivativevaluation.
     products.EuropeanOption;
 9 import net.finmath.montecarlo.process.
     EulerSchemeFromProcessModel;
10 import net.finmath.stochastic.RandomVariable;
11 import net.finmath.time.TimeDiscretization;
12 import net.finmath.time.TimeDiscretizationFromArray;
```

```
13
14 import java.util.*;
15 import java.util.stream.Collectors;
16
17 public class CurvatureRiskCalculator {
18
19     private final int numberOfPaths;
20     private final int numberOfTimeSteps;
21     private final double deltaT;
22     private final int seed;
23
24
25     public CurvatureRiskCalculator(int numberOfPaths, int
           numberOfTimeSteps, double deltaT, int seed) {
26         this.numberOfPaths = numberOfPaths;
27         this.numberOfTimeSteps = numberOfTimeSteps;
28         this.deltaT = deltaT;
29         this.seed = seed;
30     }
31
32
33     public Map<String, Map<Integer, Double[]>>
           calculateCurvatureRisk(List<Trade> trades) throws
           CalculationException {
34         Map<String, Map<Integer, Double[]>> curvatureRisk =
               new HashMap<>();
35
36         Map<String, List<Trade>> tradesByRiskFactor  = trades
               .stream()
37                 .filter(trade -> trade.getAssetType().
                       equalsIgnoreCase("Option"))
38                 .collect(Collectors.groupingBy(Trade::
                       getRiskFactorDelta));
39
40         for (Map.Entry<String, List<Trade>> entry :
               tradesByRiskFactor .entrySet()) {
41             String riskFactorDelta = entry.getKey();
42             List<Trade> riskFactorDeltaTrades = entry.
                   getValue();
43             int bucket = riskFactorDeltaTrades.get(0).
                   getBucket();
44             double riskWeight = getCurvatureRiskWeight(bucket
                   );
```

```
45
46              double CVR_plus = 0.0;
47              double CVR_minus = 0.0;
48
49              for (Trade trade : riskFactorDeltaTrades) {
50                  double delta = trade.getDelta();
51                  double shockedUnderlyingPriceUp = trade.
                        getUnderlyingPrice() * (1 + riskWeight);
52                  double shockedUnderlyingPriceDown = trade.
                        getUnderlyingPrice() * (1 - riskWeight);
53
54                  double valueBase = priceOption(trade, trade.
                        getUnderlyingPrice());
55                  double valueUp = priceOption(trade,
                        shockedUnderlyingPriceUp);
56                  double valueDown = priceOption(trade,
                        shockedUnderlyingPriceDown);
57
58                  CVR_plus += -(valueUp - valueBase -
                        riskWeight * delta);
59                  CVR_minus += -(valueDown - valueBase +
                        riskWeight * delta);
60
61                  trade.setCurvatureRisk(CVR_plus, CVR_minus);
62              }
63
64              curvatureRisk.computeIfAbsent(riskFactorDelta, k
                    -> new HashMap<>())
65                          .put(bucket, new Double[]{CVR_plus,
                                CVR_minus});
66          }
67
68          return curvatureRisk;
69      }
70
71      //pricing
72      private double priceOption(Trade trade, double
            shockedUnderlyingPrice) throws CalculationException {
73          RandomVariable initialValue = new
                RandomVariableFromDoubleArray(
                shockedUnderlyingPrice);
74          RandomVariable riskFreeRate = new
                RandomVariableFromDoubleArray(trade.
```

```
            getRiskFreeRate());
75          RandomVariable volatility = new
               RandomVariableFromDoubleArray(trade.getVolatility
               ());
76
77          BlackScholesModel model = new BlackScholesModel(
               initialValue, riskFreeRate, volatility, new
               RandomVariableFromArrayFactory());
78          TimeDiscretization timeDiscretization = new
               TimeDiscretizationFromArray(0.0, numberOfTimeSteps
               , deltaT);
79          BrownianMotion brownianMotion = new
               BrownianMotionFromMersenneRandomNumbers(
               timeDiscretization, 1, numberOfPaths, seed);
80          MonteCarloAssetModel monteCarloModel = new
               MonteCarloAssetModel(new
               EulerSchemeFromProcessModel(model, brownianMotion)
               );
81
82          RandomVariable value;
83          if (trade.getOptionStyle().equalsIgnoreCase("European
               ")) {
84              EuropeanOption europeanOption = new
                   EuropeanOption(trade.getUnderlying(), trade.
                   getMaturity(), trade.getStrikes()[0], trade.
                   getOptionType());
85              value = europeanOption.getValue(0.0,
                   monteCarloModel);
86          } else {
87              double[] exerciseDates = trade.getExerciseDates()
                   ;
88              double[] notionals = new double[exerciseDates.
                   length];
89              double[] strikes = trade.getStrikes();
90              Arrays.fill(notionals, 1.0);
91
92              BermudanOption bermudanOption = new
                   BermudanOption(exerciseDates, notionals,
                   strikes, BermudanOption.ExerciseMethod.
                   ESTIMATE_COND_EXPECTATION);
93              value = bermudanOption.getValue(0.0,
                   monteCarloModel);
94          }
```

```
95
96          return value.getAverage();
97      }
98
99      //Trova i pesi di rischio delta da utilizzare come shock
100     public static double getCurvatureRiskWeight(int bucket) {
101         Double[] weights = SensitivityAggregator.
                getRiskWeights().getOrDefault(bucket, new Double
                []{1.0, 1.0});
102
103         return weights[0];
104     }
105
106 }
```

SensitivityAggregator

```
1  package it.tesi;
2
3  import java.util.*;
4  import java.util.stream.Collectors;
5
6  public class SensitivityAggregator {
7
8      // Mappa dei pesi di rischio per ogni bucket (per delta e
           vega)
9    private static final Map<Integer, Double[]> RISK_WEIGHTS =
        new HashMap<>();
10
11   static {
12       //{Bucket, {Peso Delta, Peso Vega}}
13       RISK_WEIGHTS.put(1, new Double[]{0.55, 0.7778});
14       RISK_WEIGHTS.put(2, new Double[]{0.60, 0.7778});
15       RISK_WEIGHTS.put(3, new Double[]{0.45, 0.7778});
16       RISK_WEIGHTS.put(4, new Double[]{0.55, 0.7778});
17       RISK_WEIGHTS.put(5, new Double[]{0.30, 0.7778});
18       RISK_WEIGHTS.put(6, new Double[]{0.35, 0.7778});
19       RISK_WEIGHTS.put(7, new Double[]{0.40, 0.7778});
20       RISK_WEIGHTS.put(8, new Double[]{0.50, 0.7778});
21       RISK_WEIGHTS.put(9, new Double[]{0.70, 1.0});
22       RISK_WEIGHTS.put(10, new Double[]{0.50, 1.0});
23       RISK_WEIGHTS.put(11, new Double[]{0.70, 1.0});
```

```
24        RISK_WEIGHTS.put(12, new Double[]{0.15, 0.7778});
25        RISK_WEIGHTS.put(13, new Double[]{0.25, 0.7778});
26    }
27
28      public static Map<Integer, Double[]> getRiskWeights() {
29          return RISK_WEIGHTS;
30      }
31
32
33      //Calcola le net sensitivities delta
34      public static Map<String, Map<Integer, Double>>
           calculateNetSensitivitiesDelta(List<Trade> trades) {
35          return trades.stream()
36                  .collect(Collectors.groupingBy(
37                          Trade::getRiskFactorDelta,
38                                  Collectors.groupingBy(
39                                          Trade::getBucket,
40                                          Collectors.
                                               summingDouble(
                                               Trade::getDelta)
41                                  )
42                          )
43                  );
44      }
45
46
47      //Calcola le net sensitivities vega
48      public static Map<String, Map<Integer, Double>>
           calculateNetSensitivitiesVega(List<Trade> trades) {
49          return trades.stream()
50              .filter(trade -> trade.getAssetType() != null &&
                   !"STOCK".equalsIgnoreCase(trade.getAssetType()
                   ))
51                  .collect(Collectors.groupingBy(
52                          Trade::getRiskFactorVega,
53                                  Collectors.groupingBy(
54                                          Trade::getBucket,
55                                          Collectors.
                                               summingDouble(
                                               Trade::getVega) //
                                                Somma i valori di
                                                Vega
56                                  )
```

```
57                          )
58                 );
59     }
60
61
62      //Calcola le weighted sensitivities delta
63     public static Map<String, Map<Integer, Double>>
          calculateWeightedSensitivitiesDelta(
64               Map<String, Map<Integer, Double>>
                    netSensitivitiesDelta) {
65
66          return netSensitivitiesDelta.entrySet().stream()
67                   .collect(Collectors.toMap(
68                            Map.Entry::getKey, // RiskFactorDelta
69                            riskEntry -> riskEntry.getValue().
                               entrySet().stream()
70                                    .collect(Collectors.toMap(
71                                             Map.Entry::getKey, //
                                                 Bucket
72                                             bucketEntry -> {
73                                                  double
                                                     riskWeightDelta
                                                      =
                                                     RISK_WEIGHTS.
                                                     getOrDefault(
                                                     bucketEntry.
                                                     getKey(), new
                                                     Double[]{1.0,
                                                     1.0})[0];
74                                                  return
                                                     bucketEntry.
                                                     getValue() *
                                                     riskWeightDelta
                                                     ;
75                                             }
76                                    ))
77                 ));
78     }
79
80
81   // Calcola le weighted sensitivies vega
82   public static Map<String, Map<Integer, Double>>
        calculateWeightedSensitivitiesVega(
```

```
83        Map<String, Map<Integer, Double>> netSensitivitiesVega)
              {
84
85      return netSensitivitiesVega.entrySet().stream()
86          .collect(Collectors.toMap(
87              Map.Entry::getKey, // Risk Factor
88              riskEntry -> riskEntry.getValue().entrySet().
                    stream()
89                  .collect(Collectors.toMap(
90                      Map.Entry::getKey, // Bucket
91                      bucketEntry -> {
92                        double riskWeightVega = RISK_WEIGHTS.
                              getOrDefault(bucketEntry.getKey(),
                              new Double[] { 1.0, 1.0 })[1];
93                        return bucketEntry.getValue() *
                              riskWeightVega;
94                    }
95
96          ))));
97    }
98
99      //Trova la correlazione intra-bucket delta
100     public static double getIntraBucketCorrelation(int bucket
            , String scenario) {
101         if (bucket == 11) {
102             return 1.0;
103         }
104
105         double baseCorrelation;
106         if (bucket >= 1 && bucket <= 4) baseCorrelation =
                0.15;
107         else if (bucket >= 5 && bucket <= 8) baseCorrelation
                = 0.25;
108         else if (bucket == 9) baseCorrelation = 0.075;
109         else if (bucket == 10) baseCorrelation = 0.125;
110         else if (bucket == 11) baseCorrelation = 1.0;
111         else if (bucket == 12 || bucket == 13)
                baseCorrelation = 0.80;
112         else baseCorrelation = 0.15;
113
114         switch (scenario) {
115             case "high":
116                 return Math.min(baseCorrelation * 1.25, 1.0);
```

```
117             case "low":
118                 return Math.max(2 * baseCorrelation - 1.0,
                        0.75 * baseCorrelation);
119             default:
120                 return baseCorrelation;
121         }
122     }
123
124     //Aggregazione intra-bucket delta
125     public static Map<Integer, Double>
            aggregateIntraBucketDelta(
126             Map<String, Map<Integer, Double>>
                    weightedSensitivities, String scenario) {
127
128         Map<Integer, Double> capitalByBucket = new HashMap
                <>();
129
130         Map<Integer, List<Double>> sensitivitiesByBucket =
                new HashMap<>();
131
132         weightedSensitivities.forEach((riskFactorDelta,
                bucketMap) ->
133                 bucketMap.forEach((bucket, ws) -> {
134                     sensitivitiesByBucket.computeIfAbsent(
                            bucket, k -> new ArrayList<>()).add(ws
                            );
135                 })
136         );
137
138         for (Map.Entry<Integer, List<Double>> entry :
                sensitivitiesByBucket.entrySet()) {
139             int bucket = entry.getKey();
140             List<Double> wsList = entry.getValue();
141
142             double sumSquared = 0.0;
143             double sumCrossTerms = 0.0;
144
145             for (int i = 0; i < wsList.size(); i++) {
146                 double WS_k = wsList.get(i);
147                 sumSquared += WS_k * WS_k;
148
149                 for (int j = i + 1; j < wsList.size(); j++) {
150                     double WS_l = wsList.get(j);
```

```
151                     double correlation =
                            getIntraBucketCorrelation(bucket,
                            scenario);
152                     sumCrossTerms += correlation * WS_k *
                            WS_l;
153                 }
154             }
155
156             double capitalRequirement = Math.sqrt(Math.max(0,
                    sumSquared + sumCrossTerms));
157             capitalByBucket.merge(bucket, capitalRequirement,
                    Double::sum);
158         }
159
160         return capitalByBucket;
161     }
162
163
164     //Mappa per le maturity delle opzioni utilizzate nell'
            aggregazione intra-bucket vega
165     public static Map<String, Map<Integer, Double>>
            extractOptionMaturities(List<Trade> trades) {
166         Map<String, Map<Integer, Double>> optionMaturities =
                new HashMap<>();
167
168         for (Trade trade : trades) {
169             optionMaturities
170                 .computeIfAbsent(trade.getRiskFactorVega(), k
                        -> new HashMap<>())
171                 .put(trade.getBucket(), trade.getMaturity());
172         }
173
174         return optionMaturities;
175     }
176
177     // Aggregazione intra-bucket vega
178     public static Map<Integer, Double>
            aggregateIntraBucketVega(
179             Map<String, Map<Integer, Double>>
                    weightedSensitivities,
180             Map<String, Map<Integer, Double>>
                    optionMaturities,
181             String scenario) {
```

```
182
183         Map<Integer, Double> capitalByBucket = new HashMap
               <>();
184         Map<Integer, List<Double>> sensitivitiesByBucket =
               new HashMap<>();
185         Map<Integer, List<Double>> maturitiesByBucket = new
               HashMap<>();
186
187         weightedSensitivities.forEach((riskFactorVega,
               bucketMap) ->
188                 bucketMap.forEach((bucket, ws) -> {
189
190                         double maturity = optionMaturities.
                               getOrDefault(riskFactorVega,
                               Collections.emptyMap())
191                                 .getOrDefault(bucket, 1.0);
192
193                         sensitivitiesByBucket.computeIfAbsent
                               (bucket, k -> new ArrayList<>()).
                               add(ws);
194                         maturitiesByBucket.computeIfAbsent(
                               bucket, k -> new ArrayList<>()).
                               add(maturity);
195                 })
196         );
197
198         for (Map.Entry<Integer, List<Double>> entry :
               sensitivitiesByBucket.entrySet()) {
199             int bucket = entry.getKey();
200             List<Double> wsList = entry.getValue();
201             List<Double> maturityList = maturitiesByBucket.
                   get(bucket);
202
203             double sumSquared = 0.0;
204             double sumCrossTerms = 0.0;
205
206             for (int i = 0; i < wsList.size(); i++) {
207                 double WS_k = wsList.get(i);
208                 double T_k = maturityList.get(i);
209                 sumSquared += WS_k * WS_k;
210
211                 for (int j = i + 1; j < wsList.size(); j++) {
212                     double WS_l = wsList.get(j);
```

```
213                     double T_l = maturityList.get(j);
214
215                     double rho_delta =
                            getIntraBucketCorrelation(bucket,
                            scenario);
216                     double rho_maturity = Math.exp(-0.01 *
                            Math.abs(T_k - T_l) / Math.min(T_k,
                            T_l));
217                     double rho_kl = Math.min(rho_delta *
                            rho_maturity, 1.0);
218
219                     sumCrossTerms += rho_kl * WS_k * WS_l;
220                 }
221             }
222
223             double capitalRequirement = Math.sqrt(Math.max(0,
                    sumSquared + sumCrossTerms));
224             capitalByBucket.put(bucket, capitalRequirement);
225         }
226
227         return capitalByBucket;
228     }
229
230     //Recupera la correlazione inter-bucket per delta e vega
231   private static double getInterBucketCorrelation(int bucketA
        , int bucketB, String scenario) {
232       double baseCorrelation;
233
234       if (bucketA == 11 || bucketB == 11) return 0.0;
235
236       if (bucketA >= 1 && bucketA <= 10 && bucketB >= 1 &&
          bucketB <= 10) baseCorrelation = 0.15;
237       else if ((bucketA == 12 && bucketB == 13) || (bucketA
          == 13 && bucketB == 12)) baseCorrelation = 0.75;
238       else baseCorrelation = 0.45;
239
240       switch (scenario) {
241           case "high":
242               return Math.min(baseCorrelation * 1.25, 1.0);
243           case "low":
244               return Math.max(2 * baseCorrelation - 1.0, 0.75
                      * baseCorrelation);
245           default: // Scenario "medium"
```

```
246                 return baseCorrelation;
247         }
248     }
249
250     //Formula aggregazione inter-bucket delta e vega
251     private static double calculateInterBucketAggregation(Map<
           Integer, Double> capitalByBucket,
252         Map<String, Map<Integer, Double>> weightedSensitivities
               , String scenario) {
253       double sum_Kb2 = 0.0;
254       double sum_cross_terms = 0.0;
255
256       List<Integer> buckets = new ArrayList<>(capitalByBucket.
             keySet());
257
258       for (int i = 0; i < buckets.size(); i++) {
259         int b = buckets.get(i);
260         double K_b = capitalByBucket.getOrDefault(b, 0.0);
261         sum_Kb2 += K_b * K_b;
262
263             double S_b = weightedSensitivities.values().stream
                   ()
264                     .flatMap(bucketMap -> bucketMap.entrySet().
                        stream())
265                     .filter(entry -> entry.getKey().equals(b))
266                     .mapToDouble(Map.Entry::getValue)
267                     .sum();
268
269         if (S_b < 0) {
270           S_b = Math.max(Math.min(S_b, K_b), -K_b);
271         }
272
273         for (int j = i + 1; j < buckets.size(); j++) {
274           int c = buckets.get(j);
275           double K_c = capitalByBucket.getOrDefault(c, 0.0);
276
277                 double S_c = weightedSensitivities.values().
                      stream()
278                       .flatMap(bucketMap -> bucketMap.
                          entrySet().stream())
279                       .filter(entry -> entry.getKey().equals(
                          c))
280                       .mapToDouble(Map.Entry::getValue)
```

```
281                         .sum();
282
283         if (S_c < 0) {
284           S_c = Math.max(Math.min(S_c, K_c), -K_c);
285         }
286
287         double gamma_bc = getInterBucketCorrelation(b, c,
                scenario);
288         sum_cross_terms += gamma_bc * S_b * S_c;
289       }
290     }
291
292     return Math.sqrt(Math.max(0, sum_Kb2 + sum_cross_terms));
293   }
294
295   //Aggregazione intra-bucket delta
296   public static double aggregateInterBucketDelta(
297           Map<Integer, Double> capitalByBucket,
298           Map<String, Map<Integer, Double>>
                weightedSensitivitiesDelta,
299           String scenario) {
300
301       return calculateInterBucketAggregation(capitalByBucket,
              weightedSensitivitiesDelta, scenario);
302   }
303
304   //Aggregazione intra-bucket vega
305   public static double aggregateInterBucketVega(
306           Map<Integer, Double> capitalByBucket,
307           Map<String, Map<Integer, Double>>
                weightedSensitivitiesVega,
308           String scenario) {
309
310       return calculateInterBucketAggregation(capitalByBucket,
              weightedSensitivitiesVega, scenario);
311   }
312
313
314     //Correlazione intra-bucket curvature risk
315     private static double getAdjustedCorrelationIntra(int
          bucket, String scenario) {
316         double baseCorrelation = getIntraBucketCorrelation(
              bucket, "medium");
```

```
317         double squaredCorrelation = Math.pow(baseCorrelation,
                2);
318
319         switch (scenario) {
320             case "high":
321                 return Math.min(squaredCorrelation * 1.25,
                        1.0);
322             case "low":
323                 return Math.max(2 * squaredCorrelation - 1.0,
                        0.75 * squaredCorrelation);
324             default:
325                 return squaredCorrelation;
326         }
327     }
328
329     //Aggregazione intra-bucket curvature
330     public static Map<Integer, Double>
           aggregateCurvatureRiskIntraBucket(
331             Map<String, Map<Integer, Double[]>> curvatureRisk
                ,
332             Map<Integer, String> selectedScenarioByBucket,
333             String scenario) {
334
335         Map<Integer, Double> capitalByBucket = new HashMap
               <>();
336         Map<Integer, List<Double[]>> curvatureByBucket = new
               HashMap<>();
337
338         curvatureRisk.forEach((riskFactor, bucketMap) ->
339             bucketMap.forEach((bucket, cvrValues) ->
340                 curvatureByBucket.computeIfAbsent(bucket, k
                        -> new ArrayList<>()).add(cvrValues)
341             )
342         );
343
344         for (Map.Entry<Integer, List<Double[]>> entry :
               curvatureByBucket.entrySet()) {
345             int bucket = entry.getKey();
346             List<Double[]> cvrList = entry.getValue();
347
348             double sum_CVR_plus_squared = 0.0;
349             double sum_CVR_minus_squared = 0.0;
350             double sum_CVR_plus_corr = 0.0;
```

```
351             double sum_CVR_minus_corr = 0.0;
352
353             for (int i = 0; i < cvrList.size(); i++) {
354                 double CVR_k_plus = cvrList.get(i)[0];
355                 double CVR_k_minus = cvrList.get(i)[1];
356                 sum_CVR_plus_squared += Math.pow(Math.max(
                        CVR_k_plus, 0), 2);
357                 sum_CVR_minus_squared += Math.pow(Math.max(
                        CVR_k_minus, 0), 2);
358
359                 for (int j = i + 1; j < cvrList.size(); j++)
                        {
360                     double CVR_l_plus = cvrList.get(j)[0];
361                     double CVR_l_minus = cvrList.get(j)[1];
362                     double correlation =
                            getAdjustedCorrelationIntra(bucket,
                            scenario);
363
364                     double psi_plus = (CVR_k_plus <= 0 &&
                            CVR_l_plus <= 0) ? 0 : 1;
365                     double psi_minus = (CVR_k_minus <= 0 &&
                            CVR_l_minus <= 0) ? 0 : 1;
366
367                     sum_CVR_plus_corr += correlation *
                            CVR_k_plus * CVR_l_plus * psi_plus;
368                     sum_CVR_minus_corr += correlation *
                            CVR_k_minus * CVR_l_minus * psi_minus;
369                 }
370             }
371
372             double K_b_plus = Math.sqrt(Math.max(0,
                    sum_CVR_plus_squared + sum_CVR_plus_corr));
373             double K_b_minus = Math.sqrt(Math.max(0,
                    sum_CVR_minus_squared + sum_CVR_minus_corr));
374             double K_b = Math.max(K_b_plus, K_b_minus);
375
376             String selectedScenario;
377             if (K_b_plus == K_b_minus) {
378                 double sum_CVR_plus = cvrList.stream().
                        mapToDouble(cvr -> Math.max(cvr[0], 0)).
                        sum();
379                 double sum_CVR_minus = cvrList.stream().
                        mapToDouble(cvr -> Math.max(cvr[1], 0)).
```

```
                    sum();
380                 selectedScenario = sum_CVR_plus >
                        sum_CVR_minus ? "plus" : "minus";
381             } else {
382                 selectedScenario = (K_b == K_b_plus) ? "plus"
                        : "minus";
383             }
384
385             capitalByBucket.put(bucket, K_b);
386             selectedScenarioByBucket.put(bucket,
                    selectedScenario);
387         }
388
389         return capitalByBucket;
390     }
391
392
393     //Correlazione inter-bucket curvature
394     private static double getAdjustedCorrelationInter(int
            bucketA, int bucketB, String scenario) {
395         double baseCorrelation = getInterBucketCorrelation(
                bucketA, bucketB, "medium");
396         double squaredCorrelation = Math.pow(baseCorrelation,
                2);
397
398         switch (scenario) {
399             case "high":
400                 return Math.min(squaredCorrelation * 1.25,
                        1.0);
401             case "low":
402                 return Math.max(2 * squaredCorrelation - 1.0,
                        0.75 * squaredCorrelation);
403             default:
404                 return squaredCorrelation;
405         }
406     }
407
408     //Aggregazione inter-bucket curvature
409     public static double aggregateCurvatureRiskInterBucket(
410             Map<Integer, Double> capitalByBucket,
411             Map<Integer, String> selectedScenarioByBucket,
412             Map<String, Map<Integer, Double[]>> curvatureRisk
                    , String scenario) {
```

```
413
414         double sum_Kb2 = capitalByBucket.values().stream()
415                 .mapToDouble(K_b -> K_b * K_b)
416                 .sum();
417
418         Map<Integer, Double> S_b_map = new HashMap<>();
419         for (Map.Entry<String, Map<Integer, Double[]>> entry
                : curvatureRisk.entrySet()) {
420             for (Map.Entry<Integer, Double[]> bucketEntry :
                    entry.getValue().entrySet()) {
421                 int bucket = bucketEntry.getKey();
422                 double sum_CVR_plus = bucketEntry.getValue()
                        [0];
423                 double sum_CVR_minus = bucketEntry.getValue()
                        [1];
424
425                 String selectedScenario =
                        selectedScenarioByBucket.getOrDefault(
                        bucket, "plus");
426                 double S_b = selectedScenario.equals("plus")
                        ? sum_CVR_plus : sum_CVR_minus;
427
428                 S_b_map.merge(bucket, S_b, Double::sum);
429             }
430         }
431
432         double sum_cross_terms = 0.0;
433         List<Integer> bucketList = new ArrayList<>(S_b_map.
                keySet());
434
435         for (int i = 0; i < bucketList.size(); i++) {
436             int b = bucketList.get(i);
437             double S_b = S_b_map.getOrDefault(b, 0.0);
438
439             for (int j = i + 1; j < bucketList.size(); j++) {
440                 int c = bucketList.get(j);
441                 double S_c = S_b_map.getOrDefault(c, 0.0);
442
443                 double psi_bc = (S_b < 0 && S_c < 0) ? 0 : 1;
444                 double gamma_bc = getAdjustedCorrelationInter
                        (b, c, scenario);
445
446                 sum_cross_terms += gamma_bc * S_b * S_c *
```

```
                    psi_bc;
447             }
448         }
449
450         double result = Math.sqrt(Math.max(0, sum_Kb2 +
                sum_cross_terms));
451
452         return result;
453     }
454
455 }
```

Main

```
 1 package it.tesi;
 2
 3 import net.finmath.exception.CalculationException;
 4 import java.io.InputStream;
 5 import java.util.ArrayList;
 6 import java.util.HashMap;
 7 import java.util.List;
 8 import java.util.Map;
 9 import java.util.Set;
10
11 import java.util.TreeSet;
12
13 public class Main {
14     public static void main(String[] args) {
15
16       //Lettura csv bucket
17       InputStream bucketStream = Main.class.getClassLoader().
             getResourceAsStream("underlying_bucket_mapping.csv")
             ;
18         if (bucketStream == null) {
19             System.err.println("Errore: Il file di mappatura
                   dei bucket non    stato trovato.");
20             return;
21         }
22         CsvParser.parseUnderlyingBucketCsv(bucketStream);
23
24
25       //Lettura csv trades
```

```
26          InputStream inputStream = Main.class.getClassLoader()
               .getResourceAsStream("sample_trades.csv");
27
28          if (inputStream == null) {
29              System.err.println("Errore: Il file CSV non
                   stato trovato nella cartella resources.");
30              return;
31          }
32
33          List<Trade> trades = CsvParser.parseCsvTrade(
               inputStream);
34
35          if (trades.isEmpty()) {
36              System.out.println("Nessun trade valido trovato
                   nel file CSV.");
37              return;
38          }
39
40
41          System.out.println("=== TEST DEL PARSING DEL CSV ==="
               );
42          trades.forEach(System.out::println);
43
44          int numberOfPaths = 100000;
45          int numberOfTimeSteps = 100;
46          double deltaT = 0.1;
47          int seed = 1234;
48
49          AADPricer pricer = new AADPricer(numberOfPaths,
               numberOfTimeSteps, deltaT, seed);
50
51          try {
52              System.out.println("\n=== CALCOLO DELLE
                   SENSITIVITIES ===");
53              List<Trade> updatedTrades = pricer.
                   priceAndCalculateGreeks(trades);
54
55              Map<String, Map<Integer, Double>>
                   netSensitivitiesDelta =
56                      SensitivityAggregator.
                           calculateNetSensitivitiesDelta(
                           updatedTrades);
57              Map<String, Map<Integer, Double>>
```

```
                netSensitivitiesVega =
58                   SensitivityAggregator.
                        calculateNetSensitivitiesVega(
                        updatedTrades);
59
60              System.out.println("\n=== NET SENSITIVITIES DELTA
                     ===");
61              netSensitivitiesDelta.forEach((riskFactorDelta,
                    bucketMap) ->
62                              bucketMap.forEach((bucket, value)
                                    ->
63                                      System.out.printf("
                                         RiskFactor: %s |
                                         Bucket: %d | Net Delta
                                         : %.6f\n",
64                                           riskFactorDelta,
                                              bucket, value)
65                              ));
66
67              System.out.println("\n=== NET SENSITIVITIES VEGA
                    ===");
68              netSensitivitiesVega.forEach((riskFactorVega,
                    bucketMap) ->
69                              bucketMap.forEach((bucket, value)
                                    ->
70                                      System.out.printf("
                                         RiskFactor: %s |
                                         Bucket: %d | Net Vega:
                                          %.6f\n",
71                                           riskFactorVega,
                                              bucket, value)
72                              ));
73
74              Map<String, Map<Integer, Double>>
                    weightedSensitivitiesDelta =
75                   SensitivityAggregator.
                        calculateWeightedSensitivitiesDelta(
                        netSensitivitiesDelta);
76              Map<String, Map<Integer, Double>>
                    weightedSensitivitiesVega =
77                   SensitivityAggregator.
                        calculateWeightedSensitivitiesVega(
                        netSensitivitiesVega);
```

```
78
79              System.out.println("\n=== WEIGHTED SENSITIVITIES
                    DELTA ===");
80              weightedSensitivitiesDelta.forEach((
                    riskFactorDelta, bucketMap) ->
81                              bucketMap.forEach((bucket, value)
                                    ->
82                                      System.out.printf("
                                           RiskFactor: %s |
                                           Bucket: %d | Weighted
                                           Delta: %.6f\n",
83                                              riskFactorDelta,
                                                   bucket, value)
84                              ));
85
86              System.out.println("\n=== WEIGHTED SENSITIVITIES
                    VEGA ===");
87              weightedSensitivitiesVega.forEach((riskFactorVega
                    , bucketMap) ->
88                              bucketMap.forEach((bucket, value)
                                    ->
89                                      System.out.printf("
                                           RiskFactor: %s |
                                           Bucket: %d | Weighted
                                           Vega: %.6f\n",
90                                              riskFactorVega,
                                                   bucket, value)
91                              ));
92
93              System.out.println("\n=== CALCOLO DEL CURVATURE
                    RISK ===");
94              CurvatureRiskCalculator curvatureRiskCalculator =
                     new CurvatureRiskCalculator(numberOfPaths,
                    numberOfTimeSteps, deltaT, seed);
95              Map<String, Map<Integer, Double[]>> curvatureRisk
                     = curvatureRiskCalculator.
                    calculateCurvatureRisk(updatedTrades);
96              curvatureRisk.forEach((riskFactorDelta, bucketMap
                    ) ->
97              bucketMap.forEach((bucket, values) ->
98                      System.out.printf("RiskFactor: %s |Bucket
                           : %d | CVR +: %.6f | CVR -: %.6f\n",
99                              riskFactorDelta, bucket, values[0],
```

```
                    values[1])
100         ));
101
102         List<Trade> finalUpdatedTrades = new ArrayList
               <>();
103         for (Trade trade : updatedTrades) {
104             Double[] curvatureValues = curvatureRisk
105                     .getOrDefault(trade.
                           getRiskFactorDelta(), new HashMap
                           <>())
106                     .getOrDefault(trade.getBucket(), new
                           Double[]{0.0, 0.0});
107
108             Trade updatedTrade = new Trade(
109                     trade.getPortfolio(), trade.
                           getDealNumber(), trade.
                           getAssetType(), trade.
                           getOptionStyle(),
110                     trade.getRiskFactorDelta(), trade.
                           getRiskFactorVega(),trade.
                           getUnderlying(), trade.getBucket()
                           ,
111                     trade.getOptionType(), trade.
                           getCurrency(), trade.getAmount(),
                           trade.getVolatility(),
112                     trade.getStrikes(), trade.
                           getUnderlyingPrice(), trade.
                           getMaturity(), trade.
                           getExerciseDates(),
113                     trade.getRiskFreeRate(), trade.
                           getValue(), trade.getDelta(),
                           trade.getVega(),
114                     curvatureValues[0], curvatureValues
                           [1]
115             );
116
117             finalUpdatedTrades.add(updatedTrade);
118         }
119
120         System.out.println("\n=== TRADES AGGIORNATI CON
               DELTA, VEGA, CVR^+ e CVR^-===");
121         finalUpdatedTrades.forEach(System.out::println);
122
```

```
123
124
125             //Aggregazione intra-bucket per Delta, Vega e
                    Curvature Risk nei tre scenari
126             Map<Integer, Double> capitalByBucketDeltaMedium =
                     SensitivityAggregator.
                    aggregateIntraBucketDelta(
                    weightedSensitivitiesDelta, "medium");
127             Map<Integer, Double> capitalByBucketDeltaHigh   =
                     SensitivityAggregator.
                    aggregateIntraBucketDelta(
                    weightedSensitivitiesDelta, "high");
128             Map<Integer, Double> capitalByBucketDeltaLow    =
                     SensitivityAggregator.
                    aggregateIntraBucketDelta(
                    weightedSensitivitiesDelta, "low");
129
130             Map<String, Map<Integer, Double>>
                    optionMaturities = SensitivityAggregator.
                    extractOptionMaturities(trades);
131             Map<Integer, Double> capitalByBucketVegaMedium  =
                     SensitivityAggregator.
                    aggregateIntraBucketVega(
                    weightedSensitivitiesVega, optionMaturities, "
                    medium");
132             Map<Integer, Double> capitalByBucketVegaHigh    =
                     SensitivityAggregator.
                    aggregateIntraBucketVega(
                    weightedSensitivitiesVega, optionMaturities, "
                    high");
133             Map<Integer, Double> capitalByBucketVegaLow     =
                     SensitivityAggregator.
                    aggregateIntraBucketVega(
                    weightedSensitivitiesVega, optionMaturities, "
                    low");
134
135             Map<Integer, String> selectedScenarioByBucket =
                    new HashMap<>();
136             Map<Integer, Double>
                    capitalByBucketCurvatureMedium =
                    SensitivityAggregator.
                    aggregateCurvatureRiskIntraBucket(
                    curvatureRisk, selectedScenarioByBucket, "
```

```
            medium");
137         Map<Integer, Double> capitalByBucketCurvatureHigh
                = SensitivityAggregator.
            aggregateCurvatureRiskIntraBucket(
            curvatureRisk, selectedScenarioByBucket, "high
            ");
138         Map<Integer, Double> capitalByBucketCurvatureLow
                = SensitivityAggregator.
            aggregateCurvatureRiskIntraBucket(
            curvatureRisk, selectedScenarioByBucket, "low"
            );
139
140
141         System.out.println("\n=== AGGREGAZIONE INTRA-
            BUCKET ===");
142
143         //Delta
144         Set<Integer> allBucketsDelta = new TreeSet<>();
145         allBucketsDelta.addAll(capitalByBucketDeltaMedium
            .keySet());
146         allBucketsDelta.addAll(capitalByBucketDeltaHigh.
            keySet());
147         allBucketsDelta.addAll(capitalByBucketDeltaLow.
            keySet());
148
149         System.out.println("\n--- Delta ---");
150         for (Integer bucket : allBucketsDelta) {
151             System.out.printf("Bucket: %d | Delta (M: %.6
                f, H: %.6f, L: %.6f)\n",
152                 bucket,
153                 capitalByBucketDeltaMedium.getOrDefault(
                    bucket, 0.0),
154                 capitalByBucketDeltaHigh.getOrDefault(
                    bucket, 0.0),
155                 capitalByBucketDeltaLow.getOrDefault(
                    bucket, 0.0)
156             );
157         }
158
159         //Vega
160         Set<Integer> allBucketsVega = new TreeSet<>();
161         allBucketsVega.addAll(capitalByBucketVegaMedium.
            keySet());
```

```
162             allBucketsVega.addAll(capitalByBucketVegaHigh.
                    keySet());
163             allBucketsVega.addAll(capitalByBucketVegaLow.
                    keySet());
164
165             System.out.println("\n--- Vega ---");
166             for (Integer bucket : allBucketsVega) {
167                 System.out.printf("Bucket: %d | Vega (M: %.6f
                        , H: %.6f, L: %.6f)\n",
168                         bucket,
169                         capitalByBucketVegaMedium.getOrDefault(
                                bucket, 0.0),
170                         capitalByBucketVegaHigh.getOrDefault(
                                bucket, 0.0),
171                         capitalByBucketVegaLow.getOrDefault(
                                bucket, 0.0)
172                 );
173             }
174
175             //Curvature
176             Set<Integer> allBucketsCurvature = new TreeSet
                    <>();
177             allBucketsCurvature.addAll(
                    capitalByBucketCurvatureMedium.keySet());
178             allBucketsCurvature.addAll(
                    capitalByBucketCurvatureHigh.keySet());
179             allBucketsCurvature.addAll(
                    capitalByBucketCurvatureLow.keySet());
180
181             System.out.println("\n--- Curvature ---");
182             for (Integer bucket : allBucketsCurvature) {
183                 System.out.printf("Bucket: %d | Curvature (M:
                        %.6f, H: %.6f, L: %.6f)\n",
184                         bucket,
185                         capitalByBucketCurvatureMedium.
                                getOrDefault(bucket, 0.0),
186                         capitalByBucketCurvatureHigh.getOrDefault
                                (bucket, 0.0),
187                         capitalByBucketCurvatureLow.getOrDefault(
                                bucket, 0.0)
188                 );
189             }
190
```

```
191             //Per Delta, Vega, Curvature trovo il max intra-
                    bucket
192             Map<Integer, Double> capitalByBucketDeltaFinal =
                    new HashMap<>();
193             Map<Integer, Double> capitalByBucketVegaFinal =
                    new HashMap<>();
194             Map<Integer, Double>
                    capitalByBucketCurvatureFinal = new HashMap
                    <>();
195
196             for (Integer bucket : capitalByBucketDeltaMedium.
                    keySet()) {
197                 double maxDelta = Math.max(Math.max(
198                         capitalByBucketDeltaMedium.getOrDefault(
                                bucket, 0.0),
199                         capitalByBucketDeltaHigh.getOrDefault(
                                bucket, 0.0)),
200                         capitalByBucketDeltaLow.getOrDefault(
                                bucket, 0.0)
201                 );
202                 capitalByBucketDeltaFinal.put(bucket,
                        maxDelta);
203             }
204
205             for (Integer bucket : capitalByBucketVegaMedium.
                    keySet()) {
206                 double maxVega = Math.max(Math.max(
207                         capitalByBucketVegaMedium.getOrDefault(
                                bucket, 0.0),
208                         capitalByBucketVegaHigh.getOrDefault(
                                bucket, 0.0)),
209                         capitalByBucketVegaLow.getOrDefault(
                                bucket, 0.0)
210                 );
211                 capitalByBucketVegaFinal.put(bucket, maxVega)
                        ;
212             }
213
214             for (Integer bucket :
                    capitalByBucketCurvatureMedium.keySet()) {
215                 double maxCurvature = Math.max(Math.max(
216                         capitalByBucketCurvatureMedium.
                                getOrDefault(bucket, 0.0),
```

```
217                     capitalByBucketCurvatureHigh.getOrDefault
                            (bucket, 0.0)),
218                     capitalByBucketCurvatureLow.getOrDefault(
                            bucket, 0.0)
219                 );
220                 capitalByBucketCurvatureFinal.put(bucket,
                        maxCurvature);
221             }
222
223             //Aggregazione inter-bucket per Delta, Vega e
                    Curvature Risk nei tre scenari
224             double K_deltaMedium = SensitivityAggregator.
                    aggregateInterBucketDelta(
                    capitalByBucketDeltaFinal,
                    weightedSensitivitiesDelta, "medium");
225             double K_deltaHigh = SensitivityAggregator.
                    aggregateInterBucketDelta(
                    capitalByBucketDeltaFinal,
                    weightedSensitivitiesDelta, "high");
226             double K_deltaLow = SensitivityAggregator.
                    aggregateInterBucketDelta(
                    capitalByBucketDeltaFinal,
                    weightedSensitivitiesDelta, "low");
227
228             double K_vegaMedium = SensitivityAggregator.
                    aggregateInterBucketVega(
                    capitalByBucketVegaFinal,
                    weightedSensitivitiesVega, "medium");
229             double K_vegaHigh = SensitivityAggregator.
                    aggregateInterBucketVega(
                    capitalByBucketVegaFinal,
                    weightedSensitivitiesVega, "high");
230             double K_vegaLow = SensitivityAggregator.
                    aggregateInterBucketVega(
                    capitalByBucketVegaFinal,
                    weightedSensitivitiesVega, "low");
231
232             double K_curvatureMedium = SensitivityAggregator.
                    aggregateCurvatureRiskInterBucket(
                    capitalByBucketCurvatureFinal,
                    selectedScenarioByBucket, curvatureRisk, "
                    medium");
233             double K_curvatureHigh = SensitivityAggregator.
```

```
            aggregateCurvatureRiskInterBucket(
            capitalByBucketCurvatureFinal,
            selectedScenarioByBucket, curvatureRisk, "high
            ");
234         double K_curvatureLow = SensitivityAggregator.
            aggregateCurvatureRiskInterBucket(
            capitalByBucketCurvatureFinal,
            selectedScenarioByBucket, curvatureRisk, "low"
            );
235
236
237         //Scelta del capitale finale come massimo tra i
            tre scenari
238         double K_finalDelta = Math.max(Math.max(
            K_deltaMedium, K_deltaHigh), K_deltaLow);
239         double K_finalVega = Math.max(Math.max(
            K_vegaMedium, K_vegaHigh), K_vegaLow);
240         double K_finalCurvature = Math.max(Math.max(
            K_curvatureMedium, K_curvatureHigh),
            K_curvatureLow);
241
242         System.out.println("\n=== AGGREGAZIONE INTER-
            BUCKET ===");
243         System.out.println("\n--- Delta ---");
244         System.out.printf("(M: %.6f, H: %.6f, L: %.6f)\n"
            , K_deltaMedium, K_deltaHigh, K_deltaLow);
245         System.out.println("\n--- Vega ---");
246         System.out.printf("(M: %.6f, H: %.6f, L: %.6f)\n"
            , K_vegaMedium, K_vegaHigh, K_vegaLow);
247         System.out.println("\n--- Curvature ---");
248         System.out.printf("(M: %.6f, H: %.6f, L: %.6f)\n"
            , K_curvatureMedium, K_curvatureHigh,
            K_curvatureLow);
249
250
251         System.out.println("\n=== RISULTATI FINALI ===");
252         System.out.printf("Final Capital Requirement
            Delta: %.6f\n", K_finalDelta);
253         System.out.printf("Final Capital Requirement Vega
            : %.6f\n", K_finalVega);
254         System.out.printf("Final Capital Requirement
            Curvature: %.6f\n", K_finalCurvature);
255
```

```
256         } catch (CalculationException e) {
257             System.err.println("Errore durante il calcolo
                    delle sensitivities: " + e.getMessage());
258             e.printStackTrace();
259         }
260
261         System.out.println("\n=== FINE DEL PROGRAMMA ===");
262     }
263 }
```

Elenco delle tabelle

2.1 Classificazione dei bucket in base a: capitalizzazione di mercato, economia e settore 20
2.2 Pesi di rischio delta per i bucket da 1 a 13 21
2.3 Pesi di rischio vega per i bucket da 1 a 13 21

4.1 Descrizione dei campi del file sample_trades.csv 47
4.2 Descrizione dei campi del file underlying_bucket_mapping.csv 48
4.3 Attributi della classe Trade 51
4.4 Risk Factors Delta e Vega per ciascun trade 61