POLITECNICO DI MILANO
Corso di Laurea Magistrale di Ingegneria Matematica
Facoltà di Ingegneria dei Sistemi

Progetto di Programmazione Avanzata per il Calcolo Scientifico:

Metodo a elementi finiti per il pricing di opzioni multi-asset con modelli di Lévy

Nahuel Foresta, matr. 798775
Giorgio G. Re, matr. 799260

Anno Accademico 2012-2013

An approximate answer to the right problem is worth a good deal more than an exact answer to an approximate problem.

John Tukey

Indice

Todo list

Introduzione

Lo scopo di questo progetto è creare un piccolo *tool* scritta in c++ che possa risolvere il problema di *pricing* per alcuni derivati finanziari, tramite la soluzione di un'equazione integro-differenziale. Abbiamo quindi sviluppato la base di una libreria (molto semplice ma completamente funzionale) per il calcolo del prezzo di opzioni usando il metodo degli elementi finiti. Il tutto è basato sulla libreria ad elementi finiti esterna deal.ii, che fornisce più del necessario per la base di un programma che sfrutti il metodo degli elementi finiti sui quadrilateri. Così come la libreria deal.ii, il progetto è una libreria *template*, nel senso che quasi tutti gli oggetti utilizzati sono *templatizzati* sulla dimensione. Combinando l'uso di deal.ii per trattare la parte differenziale, e le nostre aggiunte per trattare la parte integrale, abbiamo creato una libreria che permette di calcolare il prezzo degli oggetti base (opzioni *call* e *put*) in una e due dimensioni, ma che al tempo stesso fornisce il macchinario per risolvere altri problemi simili con l'aggiunta di pochissimo codice, attraverso meccanismi di ereditarietà. Oltre al problema classico (in gergo finanziario chiamato opzione europea), abbiamo anche aggiunto la risoluzione di un problema di tipo opzione americana, in cui la soluzione deve stare sopra un dato limite, ossia un problema con ostacolo.

Una delle motivazioni che ci ha spinti a intraprendere questo progetto è la totale assenza di codici che risolvano questi problemi ad elementi finiti in più di una dimensione. Questo si spiega con il fatto che, essendo i domini quasi sempre dei rettangoli, i metodi basati sulle differenze finite, comunque più semplici da trattare, sono molto più diffusi. Da un punto di vista di prestazioni, le differenze finite non presentano particolari vantaggi sugli elementi finiti. Gli elementi finiti invece, seppur più difficili da usare, possono in alcuni casi essere più convenienti e più precisi, sopratutto per i problemi in più dimensioni. Un esempio è l'adattività locale di griglia, come vedremo in seguito.

Nei Capitolo 1 e 2 introdurremo da un punto di vista finanziario il problema, ricavando le equazioni di nostro interesse ed esplicitando le condizioni al bordo e la condizione finale. Nel Capitolo 3 invece descriveremo da un punto di vista numerico gli algoritmi e le discretizzazioni usate per risolvere i nostri problemi differenziali nei due cambi di variabile che abbiamo sfruttato. Il Capitolo 4 descrive poi gli strumenti utilizzati per la stesura del codice, quali deal.ii, CMake, GitHub. Nel capitolo 5 esporremo la struttura del nostro codice e in particolare parleremo delle classi principali che abbiamo scritto, oltre a fornire una piccola guida per un primo utilizzo della nostra libreria. Nel Capitolo 6 poi analizzeremo tramite alcuni programmi test i risultati ottenuti dai nostri codici. Nel Capitolo 7 parleremo di possibili estensioni al nostro codice, con due esempi. Infine

nel Capitolo 8 giungeremo a delle conclusioni sul lavoro fatto e in particolare sull'utilizzo del metodo a elementi finiti rispetto alle differenze finite.

Capitolo 1

Modello di Black & Scholes

1.1 Introduzione

In questo capitolo descriviamo i modelli basilari utilizzati per descrivere il mercato finanziario, seguendo le argomentazioni di Merton (1973), trattate in [2]. Consideriamo quindi un mercato finanziario molto semplificato, costituito da un titolo *risk-free* descritto dal processo B e un titolo azionario con valore pari al processo S. Definiamo quindi questi due processi.

Definizione 1.1. Sia $(\Omega, \mathcal{F}, \mu)$ uno spazio misurabile e sia $\mathcal{F}_{t\in[0,T]}$ una filtrazione. Allora, il processo B descrive il valore di un titolo *risk-free* se la sua dinamica è del tipo:

$$dB(t) = r(t)B(t)dt,$$

dove r è un qualsiasi processo $\mathcal{F}_t$-adattato.

La caratteristica più importante quindi dei processi *risk-free* è l'assenza di aleatorietà data da un processo stocastico casuale. Integrando l'equazione precedente, otteniamo:

$$B(t) = B(0)\int_0^t r(s)ds.$$

Un caso particolare è quello in cui r è una costante deterministica, in tal modo B descrive l'andamento di un'obbligazione.

Assumiamo poi che la dinamica di S sia data da:

$$dS(t) = S(t)\mu(t, S(t))dt + S(t)\sigma(t, S(t))dW(t),$$

in cui W_t è un processo di Wiener (cioè un moto browniano) e μ e σ due funzioni deterministiche. La funzione σ è detta volatilità del titolo, μ è il tasso di ritorno di S. Passiamo ora a definire il modello di *Black&Scholes*.

Definizione 1.2. Il modello di *Black&Scholes* consiste di due titoli con le seguenti dinamiche:

$$\begin{aligned} dB(t) &= rB(t)dt, \\ dS(t) &= \mu S(t)dt + \sigma S(t)dW(t), \end{aligned}$$

dove r, μ e σ sono costanti deterministiche.

1.2 Strumenti derivati e Opzioni

In questa sezione definiamo gli strumenti derivati e, in particolare, le opzioni che abbiamo trattato nel progetto.

Definizione 1.3. In finanza, è denominato strumento derivato ogni contratto o titolo il cui valore si basa sul valore di mercato di un altro titolo o strumento finanziario, detto sottostante (ad esempio, azioni, valute, tassi di interesse o derivati stessi).

Definiamo ora il titolo derivato più semplice, ovvero l'opzione *call* europea.

Definizione 1.4. Un'opzione *call* europea con prezzo di esercizio (o *strike price*) K e scadenza T sul sottostante S è un contratto finanziario derivato con le seguenti caratteristiche:

- il titolare del contratto ha, al tempo T, il diritto di acquistare un'azione del sottostante al prezzo K dal sottoscrittore del contratto, qualsiasi sia il valore del sottostante S al tempo T;
- il titolare del contratto non ha alcun obbligo di acquistare un'azione del sottostante al tempo T;
- il diritto di acquistare un'azione del sottostante può essere esercitato solo al tempo T.

Osserviamo che la scadenza del contratto e il prezzo d'esercizio sono stabiliti alla stipula del contratto, che per noi sarà tipicamente $t = 0$.
Oltre alle opzioni *call* europee, esistono opzioni *put* europee, le quali danno al titolare del contratto il diritto a vendere (anziché comprare) un dato titolo azionario a un prezzo fissato K. Le opzioni americane invece (*call* o *put* che siano), permettono di esercitare il diritto all'acquisto o alla vendita dell'azione in ogni istante di tempo $t \in [0, T]$.

Esempio 1. Supponiamo di possedere un'opzione *call* con scadenza $T = 1$ anno, *strike price* $K = 100€$ e un sottostante che al tempo $t = 0$ vale $S_0 = 100€$. Allora, se fra un anno $S_T = 120€$, eserciteremo l'opzione, acquistando il sottostante per un prezzo pari a $K = 100€$, e il sottoscrittore del contratto pagherà i rimenenti $S_T - K = 20€$. Se invece $S_T = 80€$ non eserciteremo l'opzione, ottenendo un guadagno pari a 0.

Osserviamo quindi che il valore a scadenza, cioè il *payoff*, dell'opzione dipende soltanto dal valore del sottostante. Definiamo quindi il *payoff* come variabile aleatoria in funzione di S.

Definizione 1.5. Sia S il processo stocastico che descrive l'andamento di un titolo azionario, allora il *payoff* di un'opzione scritta su S con scadenza T e *strike* K è una variabile aleatoria $\mathcal{X} \in \mathcal{F}_T$, e:

$$\mathcal{X} = \Phi(S_T).$$

Per esempio, il *payoff* delle opzioni *call* e *put* europee è:

$$\Phi_C(S_T) = \max(S_T - K, 0),$$
$$\Phi_P(S_T) = \max(K - S_T, 0).$$

La domanda che ci poniamo ora è la seguente: qual è il prezzo equo di un'opzione? Ovvero, quanto occorre pagare oggi per avere il diritto ma non l'obbligo di acquistare al tempo T un'azione a un prezzo fissato K?

1.3 L'equazione di *Black&Scholes*

Vi sono molti modi per ricavare l'equazione di *Black&Scholes*: presentiamo qui il modo più semplice e veloce.

Prima di ricavare l'equazione spendiamo qualche riga per descrivere il concetto di neutralità al rischio in finanza. Un operatore economico si dice neutrale al rischio quando le sue preferenze lo rendono indifferente al compiere un'azione il cui risultato è una quantità aleatoria, oppure compiere un'azione il cui risultato è il valore atteso della quantità aleatoria stessa.
Per esempio, per un soggetto neutrale al rischio sono indifferenti le seguenti situazioni:

- avere 1€ con probabilità 1;
- giocare a una lotteria in cui il soggetto può ricevere 2€ con probabilità $1/2$ o 0€ con probabilità $1/2$.

Nel primo caso infatti, egli ottiene sempre 1€, nel secondo ottiene in media 1€. Perciò per un soggetto neutrale al rischio queste situazioni sono indifferenti.
Quando ci occupiamo di *pricing* di derivati, ci poniamo sempre nell'ipotesi di neutralità al rischio. In particolare,

1. assumiamo che il termine di deriva μ del modello S sia pari al tasso di interesse *risk-free* r, cioè poniamo:
$$dS(t) = rS(t)dt + \sigma S(t)dW(t);$$
2. calcoliamo il valore atteso del *payoff*, cioè $\mathbb{E}\left[\Phi(S_T)|\mathcal{F}_t\right]$;
3. scontiamo il valore atteso del *payoff* con il tasso di interesse r.

Sia quindi $C : \mathbb{R}^+ \times [0,T] \to \mathbb{R}^+$, $C = C(S(t),t)$ il processo stocastico che descrive il valore di un'opzione. In particolare, se consideriamo una *call* europea,
$$C(S,t) = e^{(-r(T-t))}\mathbb{E}\left[\max(S_T - K, 0)|\mathcal{F}_t\right].$$
Data la dinamica di S, se applichiamo a questa quantità il Lemma di Itô, otteniamo la seguente equazione:
$$dC(S,t) = \left(\frac{\partial C}{\partial t} + rS\frac{\partial C}{\partial S} + \frac{1}{2}S^2\frac{\partial^2 C}{\partial S^2}\right)dt + \sigma S\frac{\partial C}{\partial S}dW(t).$$
Dall'altro lato però, per la *risk-neutrality*, la deriva di C, come quella di ogni titolo finanziario, dovrà essere pari a rC, quindi:
$$\frac{\partial C}{\partial t} + rS\frac{\partial C}{\partial S} + \frac{1}{2}S^2\frac{\partial^2 C}{\partial S^2} = rC.$$

Riassumendo, posti $C = C(S,t)$ e $P = P(S,t)$ i processi che descrivono il prezzo di opzioni *call* e *put*, otteniamo le seguenti equazioni alle derivate parziali con le rispettive condizioni finali e condizioni al bordo:

$$\begin{cases} \dfrac{\partial C}{\partial t} + rS\dfrac{\partial C}{\partial S} + \dfrac{1}{2}S^2\dfrac{\partial^2 C}{\partial S^2} = rC, \\ C(S,T) = \max(S_T - K, 0), \\ C(0,t) = 0, \qquad\qquad \forall t \in [0,T], \\ \lim\limits_{S\to\infty} C(S,t) = \infty, \qquad \forall t \in [0,T], \end{cases} \tag{1.1}$$

e

$$\begin{cases} \dfrac{\partial P}{\partial t} + rS\dfrac{\partial P}{\partial S} + \dfrac{1}{2}P^2\dfrac{\partial^2 P}{\partial S^2} = rP, \\ P(S,T) = \max(K - S_T, 0), \\ P(0,t) = K, \qquad\qquad \forall t \in [0,T] \\ \lim\limits_{S\to\infty} P(S,t) = 0, \qquad \forall t \in [0,T]. \end{cases} \tag{1.2}$$

Come possiamo osservare, si tratta di equazioni paraboliche *backward* con dato finale. Il prezzo dell'opzione sarà dato dalla soluzione C o P, valutate in $S_t = S_0$, ovvero il valore dell'azione oggi, e in $t = 0$.

1.4 Opzioni Basket

Un altro tipo di opzioni scambiate sui mercati finanziari sono le opzioni basket, il cui *payoff* dipende cioè da due o più sottostanti. In particolare, in questo progetto ci siamo concentrati sul *pricing* di opzioni basket 2D, con i seguenti valori finali:

$$C(S_1, S_2, T) = \max(S_{1,T} + S_{2,T} - K, 0)$$

per la *call* e:

$$P(S_1, S_2, T) = \max(K - S_{1,T} - S_{2,T}, 0)$$

per la *put*, dove $S_{1,T}$ e $S_{2,T}$ sono i valori al tempo T dei due sottostanti S_1 e S_2. L'equazione che si ottiene con procedimenti analoghi a quelli mostrati nella sezione precedente è la seguente:

$$\frac{\partial C}{\partial t} + rS_1\frac{\partial C}{\partial S_1} + rS_2\frac{\partial C}{\partial S_2} + \frac{\sigma_1^2}{2}S_1^2\frac{\partial^2 C}{\partial S_1^2} + \frac{\sigma_2^2}{2}S_2^2\frac{\partial^2 C}{\partial S_2^2} + \rho\sigma_1\sigma_2\frac{\partial^2 C}{\partial S_1 \partial S_2} = rC, \tag{1.3}$$

in cui $C : \mathbb{R}^+ \times \mathbb{R}^+ \times [0,T] \to \mathbb{R}^+$, $C = C(S_1(t), S_2(t), t)$, σ_1 e σ_2 sono le volatilità dei due sottostanti e ρ è il coefficiente di correlazione fra S_1 e S_2.

1.5 Opzioni Americane: il problema con frontiera libera

Le opzioni americane differiscono dalle europee poiché consentono di esercitare l'opzione non solo in T, bensì in qualsiasi istante di tempo dalla stipula del contratto alla sua scadenza. Quindi, proprio perché danno al titolare del contratto dei diritti aggiuntivi, è facile capire che vale la seguente relazione:

$$V^{Am} \geq V^{Eu},$$

ovvero il valore di un'opzione americana è sempre superiore al valore dell'europea corrispondente. In particolare, il valore dell'opzione americana è sempre pari o superiore al valore del *payoff*. L'opzione europea, infatti, può avere un valore inferiore al *payoff*, ma questo non può succedere per le americane. Infatti, se così non fosse, potremmo acquistare un'azione e una *put* su questa azione ed esercitare immediatamente, ottenendo un guadagno certo senza correre alcun rischio. Per questo valgono i seguenti vincoli:

$$C^{Am}(S,t) \geq \max(S_T - K, 0) \qquad \forall t \in [0,T], \tag{1.4}$$

$$P^{Am}(S,t) \geq \max(K - S_T, 0) \qquad \forall t \in [0,T]. \tag{1.5}$$

Consideriamo ora il comportamento della *put* nella parte sinistra del grafico

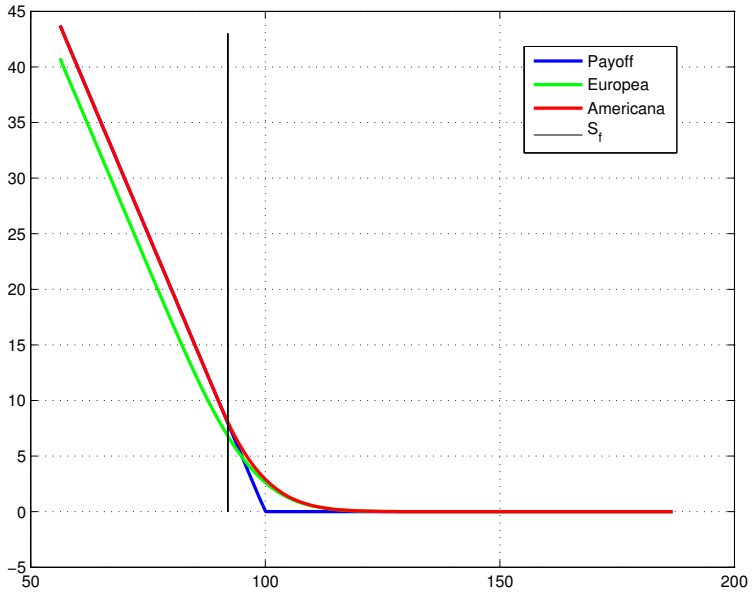

Figura 1.1: Opzioni Americane vs. Opzione Europee

riportato in figura 1.1. Senza la possibiltà di esercizio anticipato, $P^{Eu} < K - S$, ma per la disuguaglianza 1.5, $P^{Am} = K - S$. Nella parte destra della curva, invece, vale $P^{Am} \geq \max(K - S, 0)$. Quindi, per la continuità e la monotonia di P^{Am}, la curva dovrà toccare il *payoff* in un punto $S_f(t)$, $0 < S_f(t) < K$. Questo punto è definito da:

$$P^{Am}(S,t) > \max(K - S, 0) \qquad S > S_f(t),$$

$$P^{Am}(S,t) = K - S \qquad\qquad S < S_f(t).$$

Quindi, $\forall t \in [0,T]$, dobbiamo determinare il punto $S_f(t)$, attraverso il quale passa la retta che separa l'area in cui $P^{Am} =$*payoff* da qualla in cui $P^{Am} >$*payoff*. Poiché a priori questa frontiera è ignota, questo problema è detto "a frontiera

libera".

Per le *call* americane la situazione è differente, poiché $C^{Eu} \geq \max(S_T - K, 0)$[1]. Perciò il prezzo di una *call* americana è identico a quello di un'europea e in questo caso non si pone il problema con frontiera libera.

Formalmente, l'equazione differenziale che occorre risolvere per trovare il prezzo di una *put* americana è la seguente:

$$\begin{cases} \dfrac{\partial P}{\partial t} + rS\dfrac{\partial P}{\partial S} + \dfrac{1}{2}P^2\dfrac{\partial^2 P}{\partial S^2} \leq rP, \\ P(S,t) \geq \max(K - S_T, 0), \\ P(S,T) = \max(K - S_T, 0), \\ P(0,t) = K, \qquad \forall t \in [0,T] \\ \lim\limits_{S\to\infty} P(S,t) = 0, \qquad \forall t \in [0,T]. \end{cases} \tag{1.6}$$

1.6 Difetti del modello di *Black&Scholes*

Il modello di *Black&Scholes*, nonostante sia molto utilizzato in finanza, presenta alcuni problemi e può essere pericoloso utilizzarlo per valutare strumenti derivati. In particolare, empiricamente, si evidenziano le seguenti problematiche:

- il valore di S può essere discontinuo, ovvero è possibile che il sottostante presenti dei "salti";
- le code della distribuzione dei log-rendimenti dovrebbero essere normali, ma non lo sono: i valori delle code sono infatti più probabili di quanto ipotizzato dal modello, inoltre le code non sono simmetriche poiché presentano un'asimmetria verso i rendimenti negativi;
- i log-rendimenti inoltre non hanno distribuzioni indipendenti: nelle serie storiche si osservano infatti i cosiddetti *cluster*, ovvero periodi in cui la volatilità è alta e rimane alta alternati a periodi con una volatilità che rimane ridotta per lungo tempo;
- in figura 1.2 è rappresentato il grafico volatilità vs. *moneyness*[2]. In esso possiamo osservare che il valore della volatilità non rimane costante al variare dei prezzi di esercizio, bensì presenta una forma convessa. Questo fenomeno è noto come *smile* di volatilità.

Nel prossimo capitolo introdurremo una classe più ampia di processi stocastici, cioè i processi di Lévy, e descriveremo dei modelli che permettono di risolvere alcuni dei problemi sopra descritti. Questi modelli danno luogo a equazioni simili a quella di *Black&Scholes* (PDE) con l'aggiunta però di un termine integrale, per questo le chiameremo Equazioni Integro-Differenziali alle Derivate Parziali (PIDE).

[1] Questa relazione si calcola immediatamente sfruttando alcune semplici relazioni che legano C, S e K. In particolare, nel qual caso: $C^{Am} \geq C^{Eu} \geq S - Ke^{-r(T-t)} > S - K$

[2] La *moneyness* di un'opzione è il rapporto fra prezzo del sottostante in $t = 0$ e *Strike*, ovvero S_0/K.

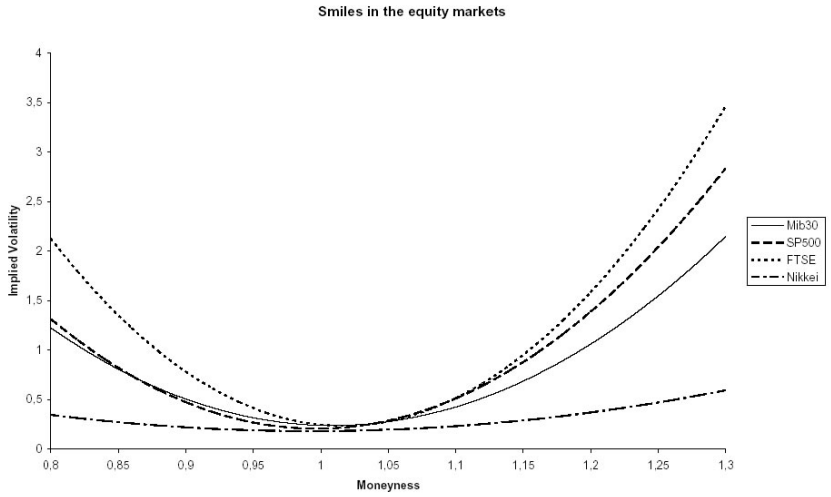

Figura 1.2: *Smile* di volatilità

Capitolo 2

Processi di Lévy e Modelli di Kou e Merton

2.1 Introduzione

In questo secondo capitolo introduciamo i processi di Lévy, una classe più ampia di processi stocastici che permettono di descrivere con più accuratezza il comportamento di un titolo azionario. Elenchiamo ora alcune definizioni e un teorema che ci permettono di definire i nuovi modelli.

Definizione 2.1. Sia $(\Omega, \mathcal{F}, \mu)$ uno spazio misurabile e sia $\mathcal{F}_{t\in[0,T]}$ una filtrazione. Sia X_t un processo stocastico *cadlag*[1], allora X_t è di Lévy se:

1. $X_0 = 0$,
2. ha incrementi indipendenti,
3. ha incrementi stazionari,
4. c'è continuità stocastica, ovvero:

$$\forall \varepsilon > 0, \qquad \lim_{h\to 0} \mathbb{P}(|X_{t+h} - X_t| \geq \varepsilon) = 0.$$

Definizione 2.2. Sia X_t un Lévy, allora poniamo

$$\nu(A) = \mathbb{E}(\#\{t \in [0,1] : \Delta X_t \neq 0, \Delta X_t \in A\}),$$

$\forall A \in \mathcal{B}$, e chiamiamo $\nu(A)$ la misura di Lévy di X_t.

Definizione 2.3. Sia X_t un processo di Lévy, allora X_t è un *Compound Poisson* di intensità λ e distribuzione di salti f se:

$$X_t = \sum_{i=1}^{N_t} Y_i,$$

dove $N_t \sim Poisson(\lambda)$ e $Y_i \sim f$, Y_i i.i.d. $\forall i$.

[1] Ricordiamo che un processo è detto *cadlag* se ha traiettorie continue a destra e limitate a sinistra.

Teorema 2.1. *Decomposizione di Lévy-Itô per processi ad attività finita.*
Sia X_t un processo di Lévy con misura ν finita, allora esistono due costanti γ e σ tali che:

$$X_t = \gamma t + \sigma W_t + X_t^C,$$

dove W_t è un moto browniano e X_t^C un Compound Poisson.

Perciò un processo di Lévy è determinato univocamente dalla sua tripletta caratteristica (γ, σ, ν).

2.2 Modelli di Merton e Kou

Passiamo ora a definire i modelli di Merton e Kou. In entrambi questi modelli il prezzo dell'azione è descritto dalla seguente equazione:

$$S_t = S_0 e^{rt+X_t}, \tag{2.1}$$

dove r è il tasso di interesse e X_t è un Lévy ad attività finita, ovvero

$$X_t = \gamma t + \sigma W_t + \sum_{i=1}^{N_t} Y_i.$$

Nel modello di Merton, $Y_i \sim \mathcal{N}(\mu, \delta^2)$, e la misura di Lévy è data da:

$$\nu(x) = \frac{\lambda}{\sqrt{2\pi\delta^2}} exp\left\{-\frac{(x-\mu)^2}{2\delta^2}\right\}, \tag{2.2}$$

in cui λ è l'intensità del *Poisson*.
Nel modello di Kou invece, le Y_i sono delle esponenziali con parametri diversi per salti positivi e negativi. In particolare,

$$\nu(x) = p\lambda\lambda_+ e^{\lambda_+ x}\mathcal{I}_{x>0} + (1-p)\lambda\lambda_- e^{-\lambda_- x}\mathcal{I}_{x<0},$$

dove p è la probabilità di salti positivi, λ è il solito parametro del *Poisson*, λ_+ e λ_- sono invece le intensità dei salti positivi e negativi.

2.3 *Pricing* con modelli Exponential Lévy

Riportiamo ora un risultato che permette di individuare un'equazione differenziale che permetta di risolvere il problema di *pricing* descritto nel capitolo precedente.

Teorema 2.2. *Sia S_t nella forma 2.1 con l'ipotesi:*

$$\int_{|x|>1} e^{2x}\nu(dx) < \infty.$$

Sia $C : \mathbb{R}^+ \times [0,T] \to \mathbb{R}^+$, $C = C(S_t, t)$ nella forma:

$$C(S_t, t) = e^{-r(T-t)}\mathbb{E}(\Phi(S_t)),$$

dove Φ *è un* payoff Lipschitz *che dipende dall'unico sottostante* S_t*. Allora* C *soddisfa l'equazione:*

$$\frac{\partial C}{\partial t} + \frac{\sigma^2}{2} S^2 \frac{\partial^2 C}{\partial S^2} + r\frac{\partial C}{\partial S} - rC + \\ + \int_{\mathbb{R}} \left(C(t, Se^y) - C(t,S) - S(e^y - 1)\frac{\partial C}{\partial S}(t,S) \right) \nu(dy) = 0. \quad (2.3)$$

Per quanto riguarda opzioni su due *asset*, nell'ipotesi che le parti di salto dei due sottostanti siano indipendenti fra di loro, l'equazione 1.3 diventa:

$$\frac{\partial C}{\partial t} + rS_1\frac{\partial C}{\partial S_1} + rS_2\frac{\partial C}{\partial S_2} + \frac{\sigma_1^2}{2}S_1^2\frac{\partial^2 C}{\partial S_1^2} + \frac{\sigma_2^2}{2}S_2^2\frac{\partial^2 C}{\partial S_2^2} + \rho\sigma_1\sigma_2 S_1 S_2 \frac{\partial^2 C}{\partial S_1 \partial S_2} - rC \\ + \int_{\mathbb{R}} \left(C(t, S_1 e^y, S_2) - C(t, S_1, S_2) - S_1(e^y - 1)\frac{\partial C}{\partial S_1}(t, S_1, S_2) \right) \nu_1(dy) \\ + \int_{\mathbb{R}} \left(C(t, S_1, S_2 e^y) - C(t, S_1, S_2) - S_2(e^y - 1)\frac{\partial C}{\partial S_2}(t, S_1, S_2) \right) \nu_2(dy) = 0, \quad (2.4)$$

dove S_1 e S_2 sono i due sottostanti descritti da modelli Exponential Lévy, rispettivamente con misure ν_1 e ν_2.
Nel prossimo capitolo, presentiamo dei metodi numerici che trovino una soluzione approssimata per le equazioni presentate in questi due capitoli.

Capitolo 3

Metodi numerici per PDE e PIDE

3.1 Introduzione

In questo capitolo descriviamo i metodi numerici utilizzati nel codice che abbiamo prodotto per approssimare le soluzioni dei problemi differenziali descritti sopra. Prima di procedere però vorremmo parlare di come sono trattate queste equazioni in finanza. Come descritto nell'introduzione infatti, generalmente, in questo campo, si utilizzano sempre metodi basati sulle differenze finite. Noi, invece, abbiamo deciso di proporre un approccio agli elementi finiti poiché riteniamo che, pur utilizzando domini per nulla complessi, i vantaggi degli elementi finiti siano evidenti anche per questo tipo di problemi, primo fra tutti la possibilità di raffinare e anche "de-raffinare" la *mesh* dove la soluzione lo richiede. Con questo tipo di approccio poi, non dovrebbe essere difficile estendere il problema al 3d, avendo cura di trattare correttamente gli integrali. Inoltre, un approccio FEM al problema 2.4 è molto difficile da trovare. In [5] si può trovare un'analisi dell'equazione in forma debole e alcuni risultati numerici (calcolati con FreeFem++), ma non vi è nessuna indicazione sugli algoritmi usati e tantomeno il codice. Per questo motivo, per validare i prezzi ottenuti, abbiamo scritto un metodo MonteCarlo per prezzare le opzioni Basket con modelli di Lévy.

Tornando ora all'argomento del capitolo, per discretizzare le equazioni abbiamo utilizzato due cambi di variabile. Il primo, che permette di portare le equazioni a coefficienti costanti, presenta l'inconveniente di dover calcolare l'integrale fuori dalla *mesh*. Il secondo invece, lascia l'equazione così com'è, ma permette di calcolare l'integrale nei soli punti della *mesh*. Lasciamo i confronti sulle prestazione dei due cambi di variabile al capitolo dedicato, tuttavia a priori è facile capire che l'assemblaggio delle matrici con il primo cambio di variabile sarà più veloce rispetto al secondo, ma il calcolo dell'integrale sarà inesorabilmente più lento.

3.2 La trasformazione *Log-Price*

Ci occupiamo ora di mostrare la discretizzazione dell'equazione con il primo cambio di variabile. Trattiamo qui il solo caso della *call* europea, in quanto per la *put* e le americane la discretizzazione è la medesima, a meno di cambiare condizioni al bordo e dati finali.

3.2.1 Equazione 1d

Il primo cambio di variabili che studiamo è il seguente: $x = log(S/S_0)$, che permette di portare l'equazione a coefficienti costanti. Posta quindi $u : \mathbb{R} \times [0,T] \to \mathbb{R}^+$, $u = u(x,t)$, incognita della PIDE monodimensionale, otteniamo:

$$\begin{cases} \dfrac{\partial u}{\partial t} + \left(r - \dfrac{\sigma^2}{2}\right)\dfrac{\partial u}{\partial x} + \dfrac{\sigma^2}{2}\dfrac{\partial^2 u}{\partial x^2} - ru \\ \qquad\qquad + \displaystyle\int_{\mathbb{R}} \left(u(t,x+y) - u(t,x) - (e^y - 1)\frac{\partial u}{\partial x}\right)\nu(dy) = 0, \\ u(x,T) = \max(S_0 e^x - K, 0), \\ \lim\limits_{x\to-\infty} u(x,t) = 0, \qquad \forall t \in [0,T], \\ \lim\limits_{x\to\infty} u(x,t) = \infty, \qquad \forall t \in [0,T], \end{cases} \tag{3.1}$$

per la *call* europea.

3.2.2 Troncamento del dominio

Come è facile osservare $x \in (-\infty, \infty)$, perciò è necessario adottare un troncamento del dominio monodimensionale. Generalmente, in finanza si applica il seguente troncamento:

$$\begin{aligned} S_{min} &= (1-f)S_0 exp\left(\left(r - \frac{\sigma^2}{2}\right)T - 6\sigma\sqrt{T}\right), \\ S_{max} &= (1+f)S_0 exp\left(\left(r - \frac{\sigma^2}{2}\right)T + 6\sigma\sqrt{T}\right), \end{aligned} \tag{3.2}$$

dove $0 \le f \le 1$ è un parametro da scegliersi a piacere (nel codice è settato a 0.5 ma è possibile modificarlo con un apposito metodo). Questo troncamento è utilizzato poiché in un *framework Black&Scholes* il sottostante S ha probabilità 10^{-8} di superare quei limiti (quando $f = 0$). Poniamo quindi:

$$x_{min} = log(S_{min}/S_0), \qquad x_{max} = log(S_{max}/S_0).$$

3.2.3 Discretizzazione della PDE

Concentriamoci ora solo sull'equazione senza parte integrale, consideriamo cioè la sola PDE del modello di *Black&Scholes*. Siano quindi Ω_h una triangolazione con $N+1$ nodi di $\Omega = [x_{min}, x_{max}]$ e $\mathcal{T} = \{0 = t_0 \le t_1 \le ... \le t_N = T\}$ una griglia temporale. Cerchiamo una soluzione del tipo:

$$u_h(x,t_n) = \sum_{j=0}^{N} \alpha_j(t_n)\phi_j(x),$$

dove α_j sono i valori della soluzione all'istante t_n nel nodo j della griglia, mentre $\phi_j(x)$ sono le funzioni base dello spazio $\mathbb{P}_h^1 = \{ f \in \mathbb{C}^0 \ \textit{lineari su ogni cella della triangolazione } \Omega_h \ \}$.
Passiamo ora alla formulazione debole del problema 1.1:

$$\sum_{j=0}^{N} \int_{\Omega_h} \left(\frac{\partial}{\partial t}(\alpha_j(t)\phi_j(x))\phi_i(x) \right) dx + \sum_{j=0}^{N} \left(r - \frac{\sigma^2}{2} \right) \int_{\Omega_h} \alpha_j(t)\phi_j'(x)\phi_i(x)dx$$
$$+ \sum_{j=0}^{N} \frac{\sigma^2}{2} \int_{\Omega_h} \alpha_j(t)\phi_j''(x)\phi_i(x)dx = \sum_{j=0}^{N} r \int_{\Omega_h} \alpha_j(t)\phi_j(x)\phi_i(x)dx, \quad (3.3)$$

$\forall t \in \mathcal{T}$ e $\forall i \in \{0, ..., N\}$. Applicando poi uno schema di Eulero Implicito per la derivata prima in tempo, giungiamo a:

$$\sum_{j=0}^{N} \int_{\Omega_h} \frac{\alpha_j(t_{n+1})\phi_j(x)}{\Delta t}\phi_i(x)dx - \sum_{j=0}^{N} \int_{\Omega_h} \frac{\alpha_j(t_n)\phi_j(x)}{\Delta t}\phi_i(x)dx$$
$$+ \sum_{j=0}^{N} \left(r - \frac{\sigma^2}{2} \right) \int_{\Omega_h} \alpha_j(t_n)\phi_j'(x)\phi_i(x)dx + \sum_{j=0}^{N} \frac{\sigma^2}{2} \int_{\Omega_h} \alpha_j(t_n)\phi_j''(x)\phi_i(x)dx$$
$$= \sum_{j=0}^{N} r \int_{\Omega_h} \alpha_j(t_n)\phi_j(x)\phi_i(x)dx, \quad (3.4)$$

$\forall t \in \mathcal{T}$ e $\forall i \in \{0, ..., N\}$. Poniamo infine:

$$a_{ij} = \int_{\Omega_h} \phi_j(x)\phi_i(x)dx,$$
$$b_{ij} = \int_{\Omega_h} \phi_j'(x)\phi_i(x)dx,$$
$$c_{ij} = \int_{\Omega_h} \phi_j'(x)\phi_i'(x)dx.$$

Otteniamo quindi la seguente discretizzazione per la PDE:

$$\sum_{j=0}^{N} \alpha_j(t_{n+1}) \frac{a_{ij}}{\Delta t} = \sum_{j=0}^{N} \alpha_j(t_n) \left(\left(\frac{1}{\Delta t} + r \right) a_{ij} - \left(r - \frac{\sigma^2}{2} \right) b_{ij} + \frac{\sigma^2}{2} c_{ij} \right).$$

3.2.4 La parte integrale

Concentriamoci ora sulla parte integrale della 3.1:

$$\int_{\mathbb{R}} \left(u(t, x+y) - u(t,x) - (e^y - 1)\frac{\partial u}{\partial x} \right) \nu(dy)$$

e osserviamo che, se la misura di probabilità è assolutamente continua rispetto alla misura di Lesbegue (e ciò è verificato per i modelli di Kou e Merton), si ha che:

$$\nu(dy) = \nu(y)dy.$$

Inoltre, siccome l'integrale della misura sullo spazio è finito nel caso dei *Compound Poisson*, possiamo separare l'integrale in tre parti distinte, e in particolare possiamo porre gli ultimi due addendi dell'integrale pari a:

$$\hat{\lambda} = \int_{\mathbb{R}} \nu(y)dy,$$
$$\hat{\alpha} = \int_{\mathbb{R}} (e^y - 1)\nu(y)dy,$$

ottenendo:

$$\frac{\partial u}{\partial t} + \left(r - \frac{\sigma^2}{2} - \hat{\alpha}\right)\frac{\partial u}{\partial x} + \frac{\sigma^2}{2}\frac{\partial^2 u}{\partial x^2} - (r + \hat{\lambda})u + \int_{\mathbb{R}} u(t, x + y)\nu(y)dy = 0.$$

Sappiamo inoltre dalla teoria che $\hat{\lambda} = \lambda$, poiché integrando la densità di Lévy otteniamo l'intensità dei salti. Per quanto riguarda invece il calcolo numerico dei due integrali, poiché le densità con cui abbiamo a che fare sono gaussiane o esponenziali, abbiamo utilizzato delle formule di quadratura rispettivamente di Hermite e di Laguerre, come spiegato in [1]. Per esempio, se consideriamo la densità di Lévy di Merton, ν è nella forma 2.2, perciò, a meno di costanti,

$$\begin{aligned}\hat{\alpha} &= \int_{\mathbb{R}} (e^y - 1)Ae^{-B(y-\mu)^2}dy \\ &\simeq A\sum_{j=0}^{Q}(e^{q_j} - 1)w_j,\end{aligned}$$

dove q_j sono i nodi di quadratura su $\mathbb{R}$, mentre w_j sono i pesi che inglobano già il nucleo guassiano. Lo stesso metodo viene utilizzato anche per il nucleo di Kou, con l'accortezza di spezzare l'integrale fra $(-\infty, 0)$ e $(0, \infty)$ poiché il parametro dell'esponenziale è diverso.

Focalizziamoci ora sul calcolo dell'ultimo integrale rimasto, cioè:

$$\int_{\mathbb{R}} u(t, x + y)\nu(y)dy, \tag{3.5}$$

e notiamo subito che dovendo valutare la funzione u nei punti $x+y$, questo è un termine non locale. Esso infatti è difficilmente trattabile poiché il termine $x+y$ introduce un possibile *shift* al di fuori dei nodi della griglia, sul quale occorre calcolare la soluzione. Per il calcolo numerico di questo termine, abbiamo deciso di utilizzare l'approccio più diffuso in finanza, ovvero un approccio simile a quello usato con i metodi alle differenze finite. In particolare, posti $x_i \in \Omega_h$ i nodi della griglia, poniamo

$$J_i = J(x_i) = \int_{\mathbb{R}} u(t, x_i + y)\nu(y)dy,$$

$\forall i \in \{0, ..., N\}$. Qualora $u(t, x_i + y)$ cada fuori dalla *mesh*, proiettiamo le condizioni al bordo del problema. Possiamo così scrivere la formulazione debole del termine 3.5 in questo modo:

$$\sum_{j=0}^{N} J_j \int_{\Omega_h} \phi_j(x)\phi_i(x)dx,$$

$\forall i \in \{0, ..., N\}$. Da un punto di vista matriciale, occorre quindi moltiplicare il vettore J per la matrice di massa i cui coefficienti sono i termini a_{ij} definiti sopra.

La discretizzazione completa in forma matriciale della PIDE monodimensionale è quindi la seguente:

$$M_1 u^n = M_2 u^{n+1} + MJ,$$

dove u^{n+1} e u^n sono la soluzione al passo precedente e successivo (ricordiamo infatti che l'equazione è *backward*) e le matrici sono date da:

$$M_{1,ij} = \left(\frac{1}{\Delta t} + r\right) a_{ij} - \left(r - \frac{\sigma^2}{2}\right) b_{ij} + \frac{\sigma^2}{2} c_{ij},$$

$$M_{2,ij} = \frac{a_{ij}}{\Delta t}, \qquad M_{ij} = a_{ij}.$$

Osserviamo quindi che l'integrale viene calcolato esplicitamente. D'altro canto, un calcolo implicito richiederebbe la costruzione di una matrice di sistema piena, molto pesante da invertire e praticamente impossibile da memorizzare su un solo computer. Dalla teoria sappiamo che questo metodo, detto *operator splitting*, è stabile se:

$$\Delta t \leq {}^1/\lambda,$$

una condizione molto semplice da soddisfare.

L'altro approccio possibile è il seguente: prima si discretizza la funzione u, scrivendola come $u_h \in \mathbb{P}_1^h$, poi si integra. Abbiamo quindi:

$$\int_{\Omega_h} \phi_i(x) \left(\int_{\mathbb{R}} \sum_{j=1}^{N} u_j^k \phi_j(x+y) \nu(y) dy \right) dx =$$

$$\sum_{j=1}^{N} \int_{\Omega_h} \int_{\mathbb{R}} \phi_i(x) \phi_j(x+y) \nu(y) dx dy,$$

che potremmo anche riscrivere, tramite un cambio di variabile prima della moltiplicazione per ϕ_i come:

$$d_{ij} = \sum_{j=1}^{N} \int_{\Omega_h} \int_{\mathbb{R}} \phi_i(x) \phi_j(z) \nu(z-x) dz dx.$$

In tal caso, potremmo porre $\{D\}_{ij} = d_{ij}$ ottenendo:

$$M_1 u^n = M_2 u^{n+1} + D u^{n+1}.$$

Questo metodo permette di calcolare una sola volta la matrice piena D, e poi moltiplicarla ogni volta per la soluzione al passo precedente (cioè $n+1$). Nonostante questa idea possa portare a un calcolo più veloce della parte integrale $J = Du^{n+1}$, anche questo metodo costringerebbe a tenere in memoria una matrice piena, quindi abbiamo scelto di scartarlo.

Siamo giunti quindi alla discretizzazione completa della PIDE 1d con la trasformazione *log-price*. Nei prossimi paragrafi mostriamo la discretizzazione della PIDE 2d.

3.2.5 Discretizzazione della PIDE bidimensionale

Trattiamo ora la PIDE bidimensionale riportata in 2.4. Tramite il cambio di variabile $x_1 = log(S_1/S_{1,0})$, $x_2 = log(S_2/S_{2,0})$, $u = u(t, x_1, x_2)$, otteniamo:

$$\frac{\partial u}{\partial t} + \frac{\sigma_1^2}{2}\frac{\partial^2 u}{\partial x_1^2} + \frac{\sigma_2^2}{2}\frac{\partial^2 u}{\partial x_2^2} + \rho\sigma_1\sigma_2\frac{\partial^2 u}{\partial x_1 \partial x_2} + \left(r - \sigma_1^2\right)\frac{\partial u}{\partial x_1} + \left(r - \sigma_2^2\right)\frac{\partial u}{\partial x_2} - ru$$
$$+ \int_{\mathbb{R}} \left(u(t, x_1 + y, x_2) - u(t, x_1, x_2) - (e^y - 1)\frac{\partial u}{\partial x_1} \right) k_1(y)dy$$
$$+ \int_{\mathbb{R}} \left(u(t, x_1, x_2 + y) - u(t, x_1, x_2) - (e^y - 1)\frac{\partial u}{\partial x_2} \right) k_2(y)dy = 0. \quad (3.6)$$

Ponendo come sopra:

$$\hat{\lambda_i} = \int_{\mathbb{R}} \nu_i(y)dy \qquad \hat{\alpha_i} = \int_{\mathbb{R}} (e^y - 1)\nu_i(y)dy,$$

abbiamo:

$$\frac{\partial u}{\partial t} + \frac{\sigma_1^2}{2}\frac{\partial^2 u}{\partial x_1^2} + \frac{\sigma_2^2}{2}\frac{\partial^2 u}{\partial x_2^2} + \rho\sigma_1\sigma_2\frac{\partial^2 u}{\partial x_1 \partial x_2} + \left(r - \frac{\sigma_1^2}{2} - \hat{\alpha_1}\right)\frac{\partial u}{\partial x_1}$$
$$+ \left(r - \frac{\sigma_2^2}{2} - \hat{\alpha_2}\right)\frac{\partial u}{\partial x_2} - (r + \hat{\lambda_1} + \hat{\lambda_2})u$$
$$+ \int_{\mathbb{R}} u(t, x_1 + y, x_2)\nu_1(y)dy + \int_{\mathbb{R}} u(t, x_1, x_2 + y)\nu_2(y)dy = 0. \quad (3.7)$$

Di nuovo, applichiamo un troncamento al dominio calcolando $\{x^1_{min}, x^1_{max}\}$ e $\{x^2_{min}, x^2_{max}\}$ come in 3.2 e definiamo una triangolazione Ω_h sul rettangolo di vertici (x^1_{min}, x^2_{min}), (x^1_{max}, x^2_{max}), la griglia temporale $\mathcal{T} = \{0 = t_0 \leq t_1 \leq ... \leq t_N = T\}$ e uno spazio $\mathbb{P}^1_h$ in cui cerchiamo una soluzione del tipo:

$$u_h = \sum_{i=0}^{N} \alpha_i(t)\phi_i(x, y),$$

con $\alpha_i(t)$ valori della soluzione nel nodo i-esimo della griglia e ϕ_i funzioni base. Prima di passare alla formulazione debole, osserviamo che l'equazione può essere scritta nel seguente modo:

$$\frac{\partial u}{\partial t} + \begin{pmatrix} r - \frac{\sigma_1^2}{2} - \hat{\alpha_1} \\ r - \frac{\sigma_2^2}{2} - \hat{\alpha_2} \end{pmatrix} \nabla u + \frac{1}{2} div\left(\begin{pmatrix} \sigma_1^2 & \rho\sigma_1\sigma_2 \\ \rho\sigma_1\sigma_2 & \sigma_2^2 \end{pmatrix} \nabla u \right) - (r + \hat{\lambda_1} + \hat{\lambda_2})u$$
$$+ \int_{\mathbb{R}} u(t, x_1 + y, x_2)\nu_1(y)dy + \int_{\mathbb{R}} u(t, x_1, x_2 + y)\nu_2(y)dy = 0. \quad (3.8)$$

Osserviamo inoltre che, pur essendo un'equazione bidimensionale, la parte integrale rimane in una sola dimensione, quindi per valutare i due integrali occorre calcolare due vettori, J_1 e J_2:

$$J_1^i = J_1(x^i) = \int_{\mathbb{R}} u(t, x_1^i + y, x_2^i)\nu_1(y)dy,$$
$$J_2^i = J_2(x^i) = \int_{\mathbb{R}} u(t, x_1^i, x_2^i + y)\nu_2(y)dy,$$

nei diversi nodi x^i della griglia bidimensionale. Siamo quindi pronti a scrivere la formulazione debole del problema (3.7):

$$\begin{aligned}
&\sum_{j=0}^{N} \iint_{\Omega_h} \frac{\partial}{\partial t} \left(\alpha_j(t)\phi_j(x,y)\phi_i(x,y)\right) d\Omega \\
&\qquad + \sum_{j=0}^{N} \begin{pmatrix} r - \frac{\sigma_1^2}{2} - \hat{\alpha_1} \\ r - \frac{\sigma_2^2}{2} - \hat{\alpha_2} \end{pmatrix} \iint_{\Omega_h} \alpha_j(t) \nabla\phi_j(x,y)\phi_i(x,y) d\Omega \\
&\quad - \frac{1}{2} \sum_{j=0}^{N} \iint_{\Omega_h} \alpha_j(t) \nabla\phi_i(x,y)^t \begin{pmatrix} \sigma_1^2 & \rho\sigma_1\sigma_2 \\ \rho\sigma_1\sigma_2 & \sigma_2^2 \end{pmatrix} \nabla\phi_j(x,y) d\Omega \\
&\qquad - (r + \hat{\lambda_1} + \hat{\lambda_2}) \sum_{j=0}^{N} \iint_{\Omega_h} \alpha_j(t)\phi_j(x,y)\phi_i(x,y) d\Omega \\
&+ \sum_{j=0}^{N} J_1^j \iint_{\Omega_h} \alpha_j(t)\phi_j(x,y)\phi_i(x,y) d\Omega + \sum_{j=0}^{N} J_2^j \iint_{\Omega_h} \alpha_j(t)\phi_j(x,y)\phi_i(x,y) d\Omega = 0,
\end{aligned} \tag{3.9}$$

e la discretizzazione in tempo con uno schema di Eulero Implicito:

$$\begin{aligned}
&\sum_{j=0}^{N} \iint_{\Omega_h} \frac{\alpha_j(t_{n+1})}{\Delta t} \phi_j(x,y)\phi_i(x,y) d\Omega \\
&+ \sum_{j=0}^{N} J_1^j \iint_{\Omega_h} \alpha_j(t_{n+1})\phi_j(x,y)\phi_i(x,y) d\Omega + \sum_{j=0}^{N} J_2^j \iint_{\Omega_h} \alpha_j(t_{n+1})\phi_j(x,y)\phi_i(x,y) d\Omega = \\
&\qquad - \sum_{j=0}^{N} \begin{pmatrix} r - \frac{\sigma_1^2}{2} - \hat{\alpha_1} \\ r - \frac{\sigma_2^2}{2} - \hat{\alpha_2} \end{pmatrix} \iint_{\Omega_h} \alpha_j(t_n) \nabla\phi_j(x,y)\phi_i(x,y) d\Omega \\
&\quad + \frac{1}{2} \sum_{j=0}^{N} \iint_{\Omega_h} \alpha_j(t_n) \nabla\phi_i(x,y)^t \begin{pmatrix} \sigma_1^2 & \rho\sigma_1\sigma_2 \\ \rho\sigma_1\sigma_2 & \sigma_2^2 \end{pmatrix} \nabla\phi_j(x,y) d\Omega \\
&\qquad + \left(\frac{1}{\Delta t} + r + \hat{\lambda_1} + \hat{\lambda_2}\right) \sum_{j=0}^{N} \iint_{\Omega_h} \alpha_j(t_n)\phi_j(x,y)\phi_i(x,y) d\Omega.
\end{aligned} \tag{3.10}$$

A questo punto, possiamo scrivere l'equazione in forma matriciale:

$$M_1 u^n = M_2 u^{n+1} + M J_1^{n+1} + M J_2^{n+1},$$

dove M_1 e M_2 sono le matrici di sistema date dalla formulazione debole e M è la matrice di massa così definite:

$$(M)_{ij} = \iint_{\Omega_h} \phi_i(x,y)\phi_j(x,y) d\Omega \qquad M_2 = \frac{1}{\Delta t} M$$

$$\begin{aligned}
(M_1)_{ij} = \iint_{\Omega_h} \Bigg[&\frac{1}{2} \nabla\phi_i(x,y)^t \begin{pmatrix} \sigma_1^2 & \rho\sigma_1\sigma_2 \\ \rho\sigma_1\sigma_2 & \sigma_2^2 \end{pmatrix} \nabla\phi_j(x,y) \\
&- \phi_i(x,y) \begin{pmatrix} r - \sigma_1^2/2 - \hat{\alpha_1} \\ r - \sigma_2^2/2 - \hat{\alpha_2} \end{pmatrix} \nabla\phi_j(x,y) + \phi_i \left(\frac{1}{\Delta t} + r + \hat{\lambda_1} + \hat{\lambda_2}\right) \phi_j \Bigg] d\Omega
\end{aligned}$$

Abbiamo dunque terminato la trattazione della trasformazione *log-price*. Nella prossima sezione vedremo il cambio di variabili *price*.

3.3 La trasformazione *Price*

Consideriamo ora il secondo cambio di variabile, un cambio solo sull'integrale. Come già anticipato, questo cambio di variabile permette di utilizzare la stessa griglia sia per l'equazione che per la quadratura dell'integrale, eliminando il problema di valutare la funzione fuori dal dominio. Inoltre risolve parzialmente il problema dello *shift* dei nodi.

3.3.1 Equazione 1d

Per semplicità, illustriamo il cambio di variabile e la relativa discretizzazione nel caso di un'equazione su un solo sottostante, per poi estenderlo al caso bidimensionale. Riportiamo l'equazione in questione:

$$\frac{\partial C}{\partial t} + \frac{\sigma^2}{2} S^2 \frac{\partial^2 C}{\partial S^2} + rS\frac{\partial C}{\partial S} - rC + \\ + \int_{\mathbb{R}} \left(C(t, Se^y) - C(t,S) - S(e^y - 1)\frac{\partial C}{\partial S}(t,S) \right) \nu(y) dy = 0.$$

Come nel caso precedente, siccome la misura di Lévy ν è finita, è possibile separare gli ultimi due addendi dentro l'integrale, e definendo:

$$\hat{\alpha} = \int_{\mathbb{R}} (e^y - 1)\nu(y)dy$$
$$\hat{\lambda} = \int_{\mathbb{R}} \nu(y)dy$$

l'equazione diventa:

$$\frac{\partial C}{\partial t} + \frac{\sigma^2}{2} S^2 \frac{\partial^2 C}{\partial S^2} + (r - \hat{\alpha})S\frac{\partial C}{\partial S} - (r + \hat{\lambda})C + \int_{\mathbb{R}} C(t, Se^y)\nu(y)dy = 0.$$

Esattamente come nel caso *log-price*, $\hat{\lambda} = \lambda$ e $\hat{\alpha}$ viene calcolato tramite una qualche formula di quadratura. A questo punto introduciamo nell'integrale il cambio di variabile:

$$z = Se^y$$

e l'equazione diventa:

$$\frac{\partial C}{\partial t} + \frac{\sigma^2}{2} S^2 \frac{\partial^2 C}{\partial S^2} + (r - \hat{\alpha})S\frac{\partial C}{\partial S} - (r + \hat{\lambda})C \\ + \int_0^{\infty} \frac{C(t,z)}{z} \nu\left(\log\left(\frac{z}{S}\right)\right) dz = 0. \quad (3.11)$$

3.3.2 Troncamento del dominio

Notiamo che il dominio della funzione e dell'integrale coincidono, ossia sono entrambi $(0, \infty)$. Scegliamo il troncamento del dominio in modo analogo al caso *log-price*, ossia definiamo S_{min} e S_{max} come in (3.2). Allo stesso modo, definiamo il dominio per l'integrale come $[S_{min}, S_{max}]$. In questo modo è possibile utilizzare la stessa griglia per il calcolo dell'integrale.

3.3.3 Discretizzazione della PDE

Concentriamoci prima sulla parte differenziale. Come nel caso *log-price*, consideriamo una triangolazione Ω_h con $N+1$ nodi di $[S_{min}, S_{max}]$. Cerchiamo una soluzione del tipo:

$$C_h(S, t_n) = \sum_{j=0}^{N} \gamma_j(t)\phi_j(S),$$

dove γ_j sono i valori della soluzione all'istante t nel nodo j della griglia, mentre $\phi_j(x)$ sono le funzioni base dello spazio $\mathbb{P}_h^1 = \{ f \in \mathbb{C}^0 \ \textit{lineari su ogni cella della triangolazione } \Omega_h \}$.
Moltiplicando l'equazione per una funzione test ϕ_i, integrando sul dominio e sostituendo la funzione approssimata C_h a C otteniamo la formulazione debole del problema (3.1):

$$\sum_{j=0}^{N} \frac{\partial}{\partial t}\gamma_j(t) \int_{\Omega_h} \phi_j(S)\phi_i(S)dS + (r - \hat{\alpha}) \sum_{j=0}^{N} \gamma_j(t) \int_{\Omega_h} S\phi_j'(S)\phi_i(S)dS$$
$$+ \frac{\sigma^2}{2} \sum_{j=0}^{N} \gamma_j(t) \int_{\Omega_h} S^2\phi_j''(S)\phi_i(S)dS - (r + \lambda) \sum_{j=0}^{N} \gamma_j(t) \int_{\Omega_h} \phi_j(S)\phi_i(S)dS = 0,$$

e integrando per parti il termine con la derivata seconda, ricordando che i termini di bordo si annullano, otteniamo:

$$\sum_{j=0}^{N} \frac{\partial}{\partial t}\gamma_j(t) \int_{\Omega_h} \phi_j(S)\phi_i(S)dS + (r - \sigma^2 - \hat{\alpha}) \sum_{j=0}^{N} \gamma_j(t) \int_{\Omega_h} S\phi_j'(S)\phi_i(S)dS$$
$$- \frac{\sigma^2}{2} \sum_{j=0}^{N} \gamma_j(t) \int_{\Omega_h} S^2\phi_j'(S)\phi_i'(S)dS - (r + \lambda) \sum_{j=0}^{N} \gamma_j(t) \int_{\Omega_h} \phi_j(S)\phi_i(S)dS = 0.$$

Questa equazione deve valere per ogni i, ossia per ogni elemento della base. Introduciamo ora una griglia temporale $\mathcal{T} = \{0 = t_0 \leq t_1 \leq ... \leq t_N = T\}$ dell'intervallo $[0, T]$. Se consideriamo uno schema temporale del tipo Eulero Implicito, con la discretizzazione nel tempo, l'equazione sopra diventa:

$$\frac{\sigma^2}{2}\sum_{j=0}^{N}\gamma_j(t_n)\int_{\Omega_h}S^2\phi_j'(S)\phi_i'(S)dS-(r-\sigma^2-\hat{\alpha})\sum_{j=0}^{N}\gamma_j(t_n)\int_{\Omega_h}S\phi_j'(S)\phi_i(S)dS$$
$$+\left(r+\lambda+\frac{1}{dt}\right)\sum_{j=0}^{N}\gamma_j(t_n)\int_{\Omega_h}\phi_j(S)\phi_i(S)dS=\sum_{j=0}^{N}\frac{\gamma_j(t_{n+1})}{dt}\int_{\Omega_h}\phi_j(S)\phi_i(S)dS$$
$$\forall i\in\{0,\dots,N\},\qquad\forall n\in\{0,\dots,T-1\}\quad(3.12)$$

Definendo il vettore γ^k come il vettore dei valori $\gamma_j(t_k)$ e le matrici seguenti

$$(M_1)_{ij}=\frac{\sigma^2}{2}\int_{\Omega_h}S^2\phi_j'(S)\phi_i'(S)dS-(r-\sigma^2-\hat{\alpha})\int_{\Omega_h}S\phi_j'(S)\phi_i(S)dS$$
$$+\left(r+\lambda+\frac{1}{dt}\right)\int_{\Omega_h}\phi_j(S)\phi_i(S)dS,$$

$$(M)_{ij}=\int_{\Omega_h}\phi_j(S)\phi_i(S)dS\quad\text{e}\quad M_2=\frac{1}{dt}M.$$

Il sistema d'equazioni si può dunque scrivere in forma matriciale come:

$$M_1\gamma^n=M_2\gamma^{n+1}\qquad\text{per } n=T-1,\dots,0\qquad(3.13)$$

3.3.4 La parte integrale

Al sistema (3.13) va aggiunta la parte integrale. Anche in questo caso viene trattata esplicitamente e aggiunta al *right hand side*. Dobbiamo dunque calcolare il seguente integrale:

$$J(S)=\int_0^\infty\frac{C(t,z)}{z}\nu\left(\log\left(\frac{z}{S}\right)\right)dz$$

che poi andrà scritto come elemento dello spazio $\mathbb{P}_h^1$. Occorre quindi calcolare l'integrale nei punti S_i della griglia, ottenendo un vettore J dove $J_i=J(S_i)$. La (3.13) diventa dunque:

$$M_1\gamma^n=M_2\gamma^{n+1}+MJ^{n+1}\qquad\text{per } n=T-1,\dots,0\qquad(3.14)$$

Il problema si riduce dunque a calcolare la seguente quantità:

$$J^n(S_i)=\int_0^\infty\frac{C(t_n,z)}{z}\nu\left(\log\left(\frac{z}{S_i}\right)\right)dz$$

per ogni nodo della griglia.
Notiamo che rispetto al metodo *log-price* non c'è più il problema dello *shift* sui nodi, ma il termine è comunque non locale perché per ogni vertice si integra su tutto il dominio.
Per il calcolo di tale termine, abbiamo deciso di adottare la seguente procedura: per ogni cella della griglia, calcoliamo il valore della funzione:

$$\frac{C(t_k,z_l)}{z}$$

su opportuni nodi di quadratura z_l. Poi, per ogni nodo S_i della griglia calcoliamo il contributo di tale cella al valore di J_i^n valutandola contro la densità calcolata in $\log(z_l/S_i)$. Riassumendo, il termine i-esimo di J^k viene calcolato come:

$$J_i^n = \sum_{Q_k \in \Omega_h} \left(\sum_{z_l \in Q_k} \frac{C(t_n, z_l)}{z_l} \nu \left(\log \left(\frac{z_l}{S_i} \right) \right) w_l \right)$$

dove Q_k sono le celle della griglia, e w_l sono i pesi di quadratura in quella cella. Notiamo che siccome la densità è calcolata in $\log(z_l/S_i)$ non occorre più fare ricorso a formule di quadratura speciali per i modelli di Kou e Merton, come nel caso *log-price*. Per questo motivo, per calcolare questi integrali abbiamo sfruttato dei classici nodi di Gauss offerti da deal.ii.

3.3.5 Discretizzazione della PIDE bidimensionale

In modo analogo è possibile trattare il caso bidimensionale. Separando le parti dell'integrale che non dipendono dal valore della funzione, possiamo riscrivere la (2.4) come:

$$\begin{aligned} \frac{\partial C}{\partial t} + (r - \hat{\alpha}_1) S_1 \frac{\partial C}{\partial S_1} + (r - \hat{\alpha}_2) S_2 \frac{\partial C}{\partial S_2} + \frac{\sigma_1^2}{2} S_1^2 \frac{\partial^2 C}{\partial S_1^2} + \frac{\sigma_2^2}{2} S_2^2 \frac{\partial^2 C}{\partial S_2^2} \\ + \rho \sigma_1 \sigma_2 S_1 S_2 \frac{\partial^2 C}{\partial S_1 \partial S_2} - (r + \lambda_1 + \lambda_2) C \\ + \int_{\mathbb{R}} C(t, S_1 e^y, S_2) \nu_1(y) dy + \int_{\mathbb{R}} C(t, S_1, S_2 e^y) \nu_2(y) dy = 0. \end{aligned}$$

Come nel paragrafo precedente, trattiamo in modo separato parte differenziale e parte integrale. Concentriamoci quindi sulla sola PDE e notiamo che essa può essere scritta in forma vettoriale, cioè:

$$\frac{\partial C}{\partial t} + \theta \cdot \nabla C + (D \nabla) \cdot \nabla C - (r + \lambda_1 + \lambda_2) C = 0 \tag{3.15}$$

dove il vettore t e la matrice D sono così definite:

$$\theta = \begin{bmatrix} (r - \hat{\alpha}^1) S_1 \\ (r - \hat{\alpha}^2) S_2 \end{bmatrix}, \qquad D = \frac{1}{2} \begin{bmatrix} \sigma_1^2 S_1^2 & \rho \sigma_1 \sigma_2 S_1 S_2 \\ \rho \sigma_1 \sigma_2 S_1 S_2 & \sigma_2^2 S_2^2 \end{bmatrix}.$$

Analogamente al caso *log-price*, il dominio va troncato seguendo lo stesso principio.
Il passaggio alla formulazione debole della parte differenziale è stata trattata in [4], ed è analoga a quella monodimensionale. In definitiva, approssimando la funzione incognita C come un elemento dello spazio $\mathbb{P}_h^1$:

$$C_h(t, S_1, S_2) = \sum_{j=0}^{N} \gamma_j(t) \phi_j(S_1, S_2),$$

si ottiene la seguente formulazione debole per la parte differenziale (omettiamo la dipendenza di ϕ_k da (S_1, S_2)):

$$\sum_{j=0}^{N}\left[\frac{\partial C}{\partial t}\gamma_j(t)\iint_{\Omega_h}\phi_i\phi_j d\Omega+\gamma_j(t)\iint_{\Omega_h}\phi_i(\theta-\mathrm{Div}\,D)\nabla\phi_j d\Omega\right.$$
$$\left.-\gamma_j(t)\iint_{\Omega_h}(\nabla\phi_i)^t D(\nabla\phi_j)d\Omega-(r-\lambda_1-\lambda_2)\gamma_j(t)\iint_{\Omega_h}\phi_i\phi_j d\Omega\right]=0$$

che deve valere per ogni elemento della base ϕ_i.
Utilizzando ancora una discretizzazione temporale del tipo Eulero Implicito otteniamo:

$$\sum_{j=0}^{N}\left[\frac{\gamma_j(t_n)}{\Delta t}\iint_{\Omega_h}\phi_i\phi_j d\Omega-\gamma_j(t_n)\iint_{\Omega_h}\phi_i(\theta-\mathrm{Div}\,D)\nabla\phi_j d\Omega\right.$$
$$\left.+\gamma_j(t_n)\iint_{\Omega_h}(\nabla\phi_i)^t D(\nabla\phi_j)d\Omega+(r-\lambda_1-\lambda_2)\gamma_j(t_n)\iint_{\Omega_h}\phi_i\phi_j d\Omega\right]=$$
$$=\sum_{j=0}^{N}\frac{\gamma_j(t_{n+1})}{\Delta t}\iint_{\Omega_h}\phi_i\phi_j d\Omega$$

Introducendo ancora una volta il vettore γ^k delle componenti di C_h al tempo k e le matrici M_1, M_2 e M:

$$(M_1)_{ij}=\left(\frac{1}{\Delta t}+r-\lambda_1-\lambda_2\right)\int_{\Omega_h}\phi_i\phi_j d\Omega+\int\int_{\Omega_h}(\nabla\phi_i)^t D(\nabla\phi_j)d\Omega$$
$$-\iint_{\Omega_h}\phi_i(\theta-\mathrm{Div}\,D)\nabla\phi_j d\Omega,$$
$$(M)_{ij}=\iint_{\Omega_h}\phi_i\phi_j d\Omega,\qquad M_2=\frac{1}{\Delta t}M,$$

possiamo scrivere il sistema in forma matriciale:

$$M_1\gamma^n=M_2\gamma^{n+1}\qquad\text{per } n=T-1,\dots,0 \tag{3.16}$$

Concentriamoci ora sulla parte integrale. Al sistema (3.16) vanno infatti aggiunte nell'*rhs* le parti integrali J_1 e J_2. Analogamente al caso monodimensionale, otteniamo:

$$M_1\gamma^n=M_2\gamma^{n+1}+MJ_1^{n+1}+MJ_2^{n+1}\qquad\text{per } n=T-1,\dots,0. \tag{3.17}$$

Per quanto riguarda il calcolo di J_1 e J_2, il caso bidimensionle è più delicato. Riportiamo le espressioni di J_1 e J_2 una volta eseguita la trasformazione *price*:

$$J_1(t,S_1,S_2)=\int_{\mathbb{R}}\frac{C(t,z,S_2)}{z}\nu_1\left(log\left(\frac{z}{S_1}\right)\right)dz,$$
$$J_2(t,S_1,S_2)=\int_{\mathbb{R}}\frac{C(t,S_1,z)}{z}\nu_1\left(log\left(\frac{z}{S_2}\right)\right)dz.$$

Per ogni punto $(S_1, S_2)_j$ della griglia, devono essere calcolati $J_1(t, S_{1,j}, S_{2,j})$ e $J_2(t, S_{1,j}, S_{2,j})$ lungo i segmenti contenuti nel dominio con direzione x e y che passano da $(S_1, S_2)_j$. Se la griglia è strutturata, questo corrisponde a integrare sulle facce delle celle. Una faccia parallela all'asse x contribuisce al calcolo dell'i-esimo termine di J_1 se la coordinata y della faccia è la stessa della coordinata y del punto $(S_1, S_2)_i$.
In sostanza, se definiamo per ogni nodo $(S_1, S_2)_j$ l'insieme $\mathcal{F}_j^1$ delle facce parallele all'asse x aventi come coordinata $y = S_{2,j}$ e in modo analogo l'insieme $\mathcal{F}_j^2$ per le facce parallele all'asse y, abbiamo le seguenti formule per il calcolo dei vettori J_1, J_2 al tempo k:

$$(J_1^k)_j = \sum_{\mathbb{F} \in \mathcal{F}_j^1} \sum_{z_l \in \mathbb{F}} \frac{C(t_k, z_l, S_2)}{z_l} \nu_1 \left(\log \left(\frac{z_l}{S_{1,j}} \right) \right) w_l,$$

$$(J_2^k)_j = \sum_{\mathbb{F} \in \mathcal{F}_j^2} \sum_{z_l \in \mathbb{F}} \frac{C(t_k, S_1, z_l)}{z_l} \nu_2 \left(\log \left(\frac{z_l}{S_{2,j}} \right) \right) w_l,$$

dove z_l, w_l sono nodi e pesi di quadratura.

3.4 Condizioni al contorno

Finora non abbiamo menzionato le condizioni al contorno. Trattandosi sempre di condizioni di tipo *dirichlet*, esse vengono imposte una volta assemblato il sistema tramite tecnica di penalizzazione. Esse differiscono leggermente per *price* e *log-price* per via della trasformazione. Osservando le equazioni che dobbiamo trattare, appare chiaro che nel caso monodimensionale le condizioni al bordo da applicare ai nostri problemi siano le seguenti:

$$C_h(S_{min}, t) = 0, \qquad C_h(S_{max}, t) = S_{max} - K,$$

per la *call* e:

$$P_h(S_{min}, t) = K - S_{min}, \qquad P_h(S_{max}, t) = 0, \tag{3.18}$$

per la *put*. Volendo essere più precisi, tuttavia, imponiamo delle condizioni al bordo con lo *Strike* K scontato, ovvero:

$$C_h(S_{min}, t) = 0, \qquad C_h(S_{max}, t) = S_{max} - Ke^{-r(T-t)},$$

per la *call* e:

$$P_h(S_{min}, t) = Ke^{-r(T-t)} - S_{min}, \qquad P_h(S_{max}, t) = 0,$$

per la *put*. Questo sconto viene fatto per riportare la quantità monetaria K al tempo t adattandola tramite il tasso d'interesse *risk-free*, in modo da preservare i principi di non arbitraggio. Sarebbe infatti leggermente errato considerare costante nel tempo il valore di un oggetto monetario senza rischio, visto che anch'esso deve rispettare la logica del non arbitraggio.
Nel caso bidimensionale occorre quindi imporre per ogni nodi di bordo S_i:

$$C_h(S_i, t) = \max \left(S_{max}^1 + S_{max}^2 - Ke^{-r(T-t)}, 0 \right)$$

per la *call* e:

$$P_h(S_i,t) = \max\left(Ke^{-r(T-t)} - S^1_{min} - S^2_{min}, 0\right),$$

per la *put*.
Chiaramente, quando si fa un cambio di variabili per passare al caso *log-price*, è necessario cambiare le condizioni al contorno scrivendole nella nuova variabile. Per esempio per una *call*:

$$u_h(x_{min},t) = 0, \qquad u_h(x_{max},t) = S_0 e^{x_{max}} - Ke^{-r(T-t)}$$

Analogamente si definiscono le condizioni al bordo per *put* e per il caso bidimensionale.

3.5 Il problema con l'ostacolo: il SOR proiettato

Il metodo più utilizzato in finanza per risolvere problemi del tipo (1.6) è il cosiddetto SOR proiettato, un metodo di risoluzione del sistema lineare iterativo che permette ad ogni iterazione e per ogni punto della griglia di imporre che la soluzione stia sopra all'ostacolo. L'algoritmo, molto semplice da implementare, è una variazione del noto metodo SOR, *successive over-relaxation*, derivato dal metodo di *Gauss-Seidel*. Posto quindi:

$$z = \frac{1}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij}x_j^{(k+1)} - \sum_{j>i} a_{ij}x_j^{(k)}\right),$$

dove b è il termine noto, a è la matrice e $x^{(k)}$ e $x^{(k+1)}$ sono le soluzione al passo precedente e successivo, al posto della classica iterata $(k+1)$-esima data da:

$$x_i^{(k+1)} = x_i^{(k)} + \omega(z - x_i^{(k)}),$$

con $0 < \omega < 2$ fissato, nel SOR proiettato imponiamo:

$$x_i^{(k+1)} = \max\left(K - S_i, \quad x_i^{(k)} + \omega(z - x_i^{(k)})\right),$$

dove S_i è il nodo i-esimo della griglia. In questo modo, qualora la soluzione scenda sotto al *payoff*, imponiamo che essa valga esattamente il *payoff* rispettando il vincolo (1.5).
In questo problema, occorre prestare particolare attenzione alle condizioni al bordo: sebbene la forma sia la stessa, in questo caso non dobbiamo scontare la quantità K. Infatti, proprio perché l'opzione può essere esercitata a ogni istante temporale $t \in [0,T]$, il guadagno che si ottiene è pari a $K - S_t$, calcolato usando direttamente lo *strike* K. Perciò in questo caso la condizione al bordo da imporre è nella forma 3.18, ovvero con il prezzo di esercizio non scontato.

3.6 *Mesh refinement*

In questo programma abbiamo implementato anche una funzione che permette di adattare la griglia al problema, in particolare di raffinare o de-raffinare la *mesh*

dove ce ne sia bisogno. Per capire in quali punti adattare la griglia abbiamo utilizzato lo stimatore di Kelly, implementato direttamente nella libreria deal.ii. Esso stima l'errore soltanto per l'equazione generalizzata di Poisson:

$$-\nabla\left(a(x)\nabla u\right) = f,$$

tuttavia è uno stimatore utilizzato spesso perché permette di stimare la differenza fra i gradienti della soluzione in due celle vicine. Qualora questa differenza sia troppo grande, ovvero la soluzione cambi molto inclinazione fra celle vicine, allora è necessario raffinare la griglia. Se invece la differenza fra i gradienti è nulla o molto piccola, allora è possibile de-raffinare la *mesh*.
A causa dell'algoritmo usato per il calcolo delle parti integrali, il *mesh refinement* non è compatibile con la trasformazione *price* nel caso bidiminesionale. Infatti, se per esempio una cella venisse raffinata, si verrebbe a creare un nodo che appartiene a un segmento lungo l'asse x (e a uno lungo l'asse y) ma che esiste solo all'interno della cella raffinata, e non su tutto il dominio. In tal caso, a quel nodo si sommerebbero i contributi solo di quella cella e non di tutto $[S_{min}, S_{max}]$, il che porterebbe a un errore concettuale. Per questo motivo, il *mesh refinement* della PIDE bidimensionale in *price* è disattivato. Nella trasformazione *log-price* non vi sono invece problemi, dato che il metodo implementato funziona su una griglia qualunque.

Capitolo 4

Pacchetti usati

4.1 deal.ii

deal.ii è una libreria scritta in c++ che permette di risolvere tramite metodi a elementi finiti per una grande varietà di problemi alle derivate parziali. Fra i suoi maggiori pregi, oltre alla presenza di elementi finiti di ogni ordine, troviamo la grande scalabilità (è stato infatti provato che questa libreria riesce a scalare fino a 16.000 processori), la possibilità di interfacciarsi con molte librerie come PETSc, Trilinos, METIS, BLAS, LAPACK, e la vastità e la precisione della documentazione. Soffermandoci un attimo su questo ultimo punto, vorremmo proprio sottolineare la presenza di una documentazione (quasi) sempre precisa e ben organizzata. Oltre a questa, ci sono inoltre 51 tutorial che insegnano a utilizzare la libreria dalla semplice creazione di una *mesh* bidimensionale fino alla risoluzione di problemi iperbolici di meccanica non lineare in 3d con *mesh refinement*. Per scrivere il nostro codice abbiamo sfruttato molto la massiccia documentazione della libreria, e siamo stati quasi sempre in grado di trovare le funzioni che facevano al caso nostro. In una occasione tuttavia non siamo riusciti a trovare un metodo e abbiamo deciso di scrivere sul gruppo `https://groups.google.com/forum/#!forum/dealii`, ottenendo una risposta precisa in breve tempo.
In definitiva, la libreria deal.ii si presenta come un prodotto solido e con una curva di apprendimento molto morbida all'inizio, ma che necessita un po' piú di sforzo se si vogliono utilizzare funzionalità meno comuni.

4.2 Cmake

La libreria deal.ii utilizza Cmake per compilare i codici dei tutorial. Perciò anche noi abbiamo utilizzato questo *tool*. In particolare, in un qualsiasi file CMakeLists.txt come quelli forniti da deal.ii occorre aggiungere tutti i file sorgente, scrivere il nome dell'eseguibile desiderato e il CMake crea da solo il Makefile specificando dove si trovano gli *header file* da includere e le librerie di deal.ii da linkare. Utilizzando due sistemi operativi diversi, il CMake è stato molto utile per poter lavorare sui codici senza avere problemi di portabilità che può avere il semplice Makefile. Inoltre, siccome abbiamo utilizzato nel nostro codice

OpenMP e il CMakeLists.txt di deal.ii non lo prevede, abbiamo aggiunto la ricerca del pacchetto OpenMP nel modo seguente:

```
find_package(OpenMP)
if (OPENMP_FOUND)
    set (CMAKE_C_FLAGS "${CMAKE_C_FLAGS}
        ${OpenMP_C_FLAGS}")
    set (CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS}
        ${OpenMP_CXX_FLAGS}")
endif()
```

in modo che, se il CMake trova OpenMP, aggiunge il *flag* -fopenmp al compilatore, altrimenti non aggiunge nulla.

4.3 GitHub

Durante lo sviluppo di questo progetto, abbiamo sempre utilizzato il sistema di controllo di versioni Git e il servizio di *web hosting* GitHub per lo sviluppo di progetti software. Abbiamo quindi creato una *repository* remota all'indirizzo `https://github.com/NTFrs/pacs_proj`. L'uso di git ci ha permesso di lavorare in contemporanea sul progetto senza doverci preoccupare di fare cambiamenti allo stesso file, anche attraverso l'uso di diversi *branch*.

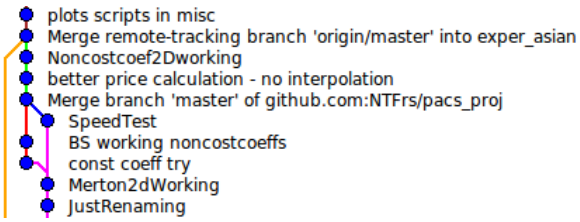

Figura 4.1: Cammino di alcuni *branch* sui quali abbiamo lavorato. Immagine tratta da gitk.

4.4 Profilers e Memory Checkers: gprof e valgrind

Durante la scrittura dei codici finali, abbiamo notato che alcune parti del programma erano più lente rispetto a implementazioni fatte precedentemente. L'utilizzo del *profiler* gprof ci ha permesso di analizzare e ristrutturare quelle parti ottenendo uno *speedup* considerevole, anche rispetto ai codici antecedenti. In particolare, il calcolo dell'integrale con la trasformazione *price* non era molto veloce, e tramite il *profiler* abbiamo scoperto che una delle funzioni di deal.ii di accesso agli elementi della griglia era particolarmente lenta. Abbiamo quindi cercato di ridurre quanto possibile il numero di chiamate ad essa.
Abbiamo anche analizzato il codice con l'uso di valgrind e usando i seguenti *tool*:

- callgrind: nel caso *log-price*, eravamo interessati a sapere quanto la parte compute_J() incidesse sul programma per valutare l'effetto della parallelizzazione. Purtroppo, gprof non riesce a riconoscere bene la successione di chiamate di funzioni (in particolare alcune interne a deal.ii) e quindi abbiamo utilizzato il tool callgrind di valgrind insieme al visualizzatore KCacheGrind (o QCacheGrind), ottenendo un'analisi migliore. La figura 4.2 riporta la parte del *call graph* a cui eravamo interessati. Fra i programmi forniti si può trovare profiled che implementa il test, così come il file ottenuto in seguito al profiling.

- memcheck: abbiamo inoltre analizzato i nostri programmi alla ricerca di *memory leaks*, notando un comportamento un po' strano di valgrind: l'output del programma afferma che non ci sono problemi di memoria, ma è possibile che ci sia un *leak* di 792 byte. Questo valore rimane costante in ogni programma testato, anche allocando due o più oggetti Opzione, e la lista di chiamate a funzione stampata da valgrind indica che il problema si trova all'interno della funzione che esegue il prodotto matrice-vettore di deal.ii. Per questo motivo riteniamo che valgrind non capisca cosa succeda in quella funzione e ritenga che ci possa essere un *memory leak*. A scanso di equivoci, abbiamo anche analizzato i programmi con un *tool* di Xcode che controlla l'uso della memoria, e quest'ultimo non ha evidenziato alcun problema.

4.5 Doxygen

Tutto il codice è commentato con il *lexical-scanner* Doxygen, molto semplice da utilizzare.

4.6 Quadrature Rules

Per il calcolo degli integrali con nodi di Laguerre e Hermite, abbiamo utilizzato una piccola libreria di funzioni utilizzata durante uno dei laboratori del corso, disponibile sul sito `http://people.sc.fsu.edu/~jburkardt/cpp_src/laguerre_rule/laguerre_rule.cpp` e distribuita sotto la licenza GNU LGPL. Si tratta di un insieme di funzioni scritte in C che permettono di calcolare nodi di integrazione di diverso tipo: oltre a quelli di nostro interesse, Chebyshev, Jacobi, esponenziali, razionali e altri.

4.7 astyle

Abbiamo infine utilizzato un piccolo programma disponibile al sito `http://astyle.sourceforge.net/` che permette di formattare automaticamente il codice, in modo da avere un *layout* uniforme in tutti i file sorgente.

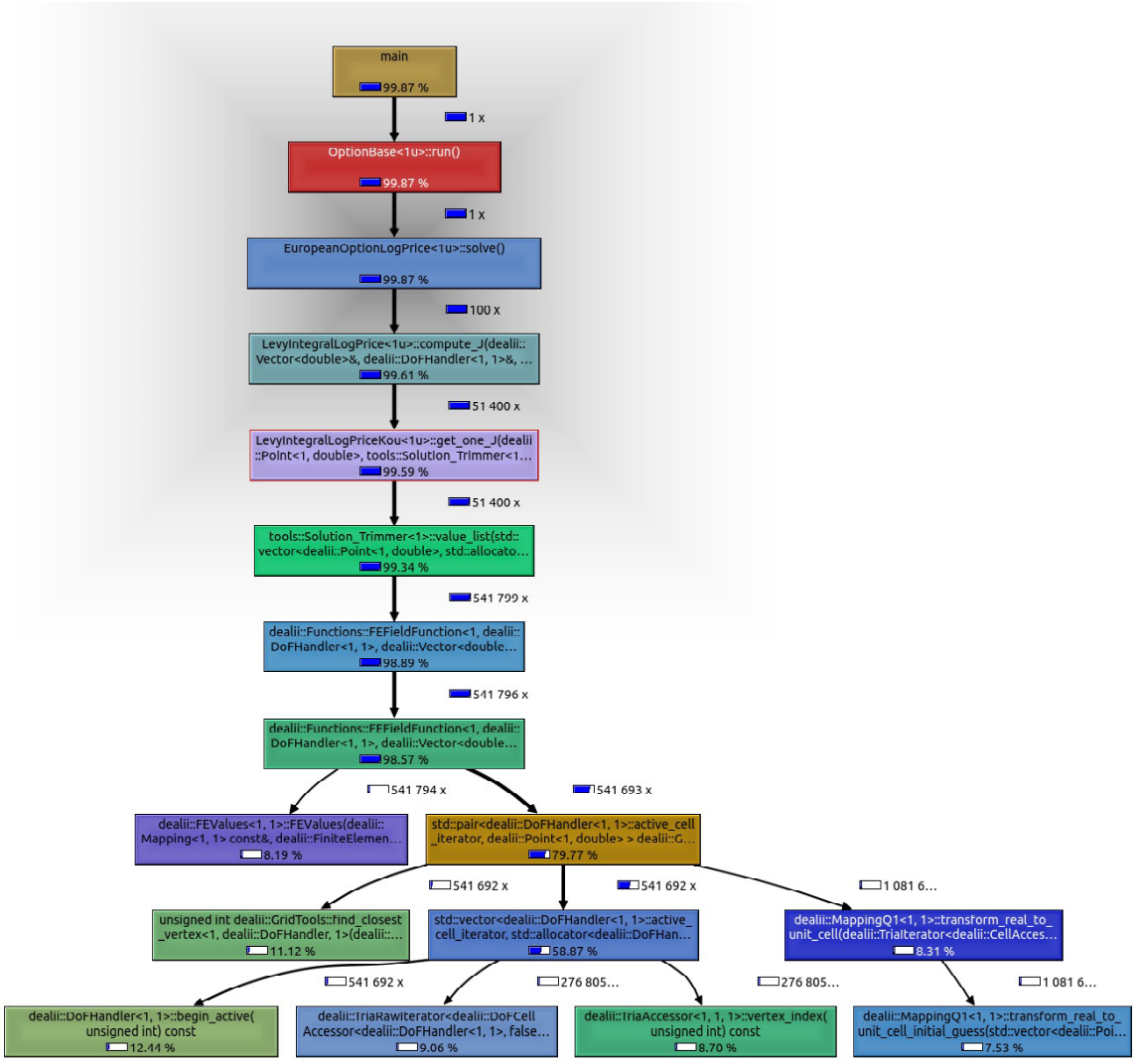

Figura 4.2: Call Graph generato da KCacheGrind con i pesi relativi delle funzioni per un'opzione con trasformazione *log-price* monodimensionale. Le funzioni con un costo relativo basso non sono mostrate.

Capitolo 5

Codice

5.1 Introduzione

In questo capitolo descriviamo come il problema è stato implementato da un punto di vista computazionale, elencando le varie classi scritte e le loro caratteristiche. Inizialmente mostriamo delle piccole classi che si occupano di gestire i parametri dei modelli e le loro densità. Nella seconda sezione descriviamo gli oggetti opzione che costruiscono il sistema per risolvere il problema a elementi finiti in una e due dimensioni. Questi oggetti utilizzano poi le funzioni di altri oggetti, LevyIntegral, per il calcolo della parte integrale. Successivamente, descriviamo brevemente la *Factory* che abbiamo scritto per instanziare gli oggetti opzione e infine spieghiamo brevemente le linee guida per utilizzare questa libreria.

5.2 Classi per i Modelli

Il primo blocco di classi scritto nel nostro codice permette di gestire i vari modelli utilizzati per descrivere la dinamica del sottostante, in particolare i modelli di *Black&Scholes*, *Kou* e *Merton*. Abbiamo quindi deciso di scrivere una classe astratta Model, che contenga i parametri comuni ai diversi oggetti e tutti i metodi utilizzati. Come possiamo osservare in figura 5.2, da questa classe ereditano in maniera pubblica le tre classi modello non più astratte, ed esse contengono i parametri propri di ogni modello.

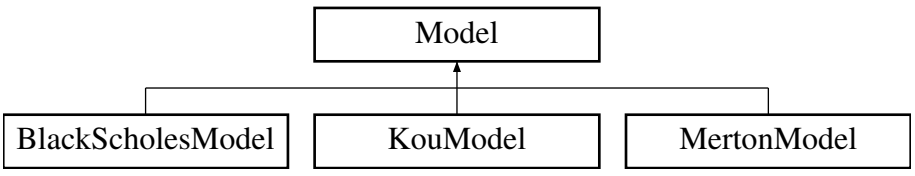

Figura 5.1: Gerarchia delle classi modello

5.3 Classi per Opzioni

Le classi per Opzioni sono la componente centrale del programma e permettono di istanziare e gestire il problema di differenziale. Sono state scritte secondo le linee guida della libreria deal.ii. Le loro caratteristiche principali sono dunque le seguenti:

- come tutte le classi presentate nei *tutorial* di deal.ii, anche le nostre hanno la dimensione (nel nostro caso 1 o 2) come parametro *template*
- i più importanti metodi privati (o meglio, protetti) e pubblici dell'oggetto Opzione sono i seguenti:
  1. virtual void setup_system();
  2. virtual void make_grid();
  3. virtual void assemble_system();
  4. virtual void refine_grid();
  5. virtual void setup_integral(), aggiunto da noi per allocare le classi che si occupano della parte integrale;
  6. virtual void solve(), metodo che risolve il sistema lineare;
  7. virtual void run(), metodo pubblico che chiama in ordine tutti i precedenti e calcola la soluzione.

A questi metodi sono state aggiunte altre funzioni che permettono di impostare alcune variabili del problema.
Per quanto riguarda la struttura gerarchica di queste classi, poiché il nostro scopo è calcolare il prezzo di opzioni europee e americane con trasformazioni *price* e *log-price* e poiché alcuni metodi sono comuni ai diversi problemi e alle diverse trasformazioni, abbiamo deciso di sfruttare l'ereditarietà delle classi permessa dal c++ facendo chiamare al programma per ogni problema i metodi opportuni.

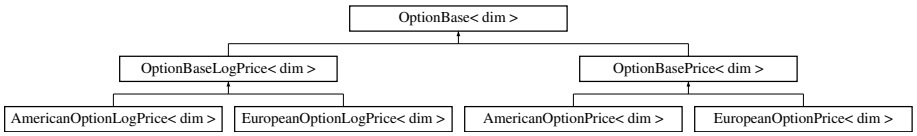

Figura 5.2: Gerarchia delle classi opzione

5.3.1 OptionBase<dim>

Entrando più nel dettaglio, come possiamo osservare nella figura 5.2, abbiamo scritto una classe base OptionBase<dim> astratta, nel cui campo protected sono contenute tutte le variabili del problema, quali gli oggetti dealii::FE_Q<dim>, dealii::DoFHandler<dim> e dealii::Triangulation<dim> che gestiscono elementi finiti, gradi di libertà e *mesh*, gli oggetti che memorizzano le matrici e i

dealii::Vector<double> della soluzione e del *right hand side*. Per quanto riguarda l'oggetto che gestisce la matrice di sistema, poiché uno dei nostri problemi necessita di un *solver* particolare non presente nella libreria utilizzata, ovvero il PSOR, abbiamo deciso di scrivere un piccolo decoratore. Questo oggetto, dealii::SparseMatrix_PSOR<double, dim> eredita pubblicamente da dealii::SparseMatrix<double> e aggiunge un metodo che risolva il problema con l'ostacolo. Come sappiamo, i costruttori non vengono ereditati, perciò sono stati riscritti in modo che chiamino i costruttori della classe base. Inoltre, siccome i metodi di dealii::SparseMatrix<double> non sono virtual, abbiamo aggiunto all'interno della nostra classe:

```
// SparseMatrix methods
using dealii::SparseMatrix<number>::reinit;
using dealii::SparseMatrix<number>::add;
```

in modo che il compilatore capisca che deve utilizzare i metodi reinit e add di dealii::SparseMatrix<double>.

Per quanto riguarda invece i costruttori della classe Opzione, essi sono specializzati per 1 e 2 dimensioni. In particolare, il costruttore 1d prende un puntatore alla classe base Model e i vari parametri dell'opzione (quali tasso di interesse, scadenza e *strike*), mentre il costruttore 2d prende due puntatori a due oggetti Model e i vari dati dell'opzione. I costruttori si occupano di creare gli oggetti relativi agli elementi finiti e li collegano alla *mesh*. I metodi invece che implementiamo qui sono void setup_system() che si occupa di creare lo *sparsity pattern* delle matrici e inizializzarle e void refine_grid() che, tramite lo stimatore di *Kelly*, esegue un adattamento della griglia.

5.3.2 OptionBasePrice<dim> e OptionBaseLogPrice<dim>

Le classi OptionBasePrice<dim> e OptionBaseLogPrice<dim>, anch'esse astratte, ereditano da OptionBase<dim> e definisco parte dei metodi descritti nell'introduzione. Le due classi implementano la funzione void make_grid() che crea la *mesh* nei casi *price* e *log-price*, ovvero con le opportune trasformazioni, il metodo void assemble_system() che integra gli elementi finiti e costruisce le matrici di sistema in 1d e 2d e il metodo double get_price(), che valuta la soluzione nel punto $x = (0, 0)$ per il *log-price* e $\underline{S} = (S_0^1, S_0^2)$ per il *price*, ovvero che restituisce il prezzo dell'opzione. Infine, la classe OptionBasePrice<dim> implementa qui il metodo void setup_integral() che istanzia dinamicamente un oggetto di tipo LevyIntegral, che si occupa di calcolare la parte integrale. OptionBaseLogPrice<dim> invece non lo istanzia qui ma nel "livello" di ereditarierà successivo poiché con questa trasformazione occorre conoscere le condizioni al bordo per poter calcolare l'integrale.

5.3.3 AmericanOption<dim> e EuropeanOption<dim>

Queste quattro classi alla base della piramide sono gli oggetti utilizzati per risolvere i vari problemi. Essi implementano le varie funzioni void solve() che risolvono il sistema lineare. In primo luogo quindi viene proiettata sulla *mesh* la condizione finale (ovviamente diversa fra *put* e *call*, *price* e *log-price*) e successivamente parte il ciclo temporale che applica le condizioni al bordo, calcola il vet-

tore integral J se il modello non è *Black&Scholes* e risolve il sistema. Per quanto riguarda quest'ultimo passaggio, per l'europea abbiamo utilizzato un *solver* usato da deal.ii, ovvero SparseDirectUMFPACK, mentre per l'americana usiamo il *solver* scritto all'interno del decoratore di dealii::SparseMatrix<double>.

5.4 Classi per il calcolo degli integrali

Per il calcolo della parte integrale dell'equazione, ossia la quadratura di $\hat{\alpha}$ e il calcolo dei vettori J_i ad ogni iterazione temporale, sono state costruite una serie di classi. Tali classi ereditano da una classe comune di base che definisce l'interfaccia. Usando questo schema con ereditarietà (mostrato in figura 5.3) siamo riusciti a mantenere un'interfaccia comune, pur allocando classi specifiche per modelli specifici. Così come le classi opzione, anche queste classi sono *templatizzate* sulla dimensione, poiché il calcolo dell'integrale può essere anche molto diverso fra una e due dimensioni.

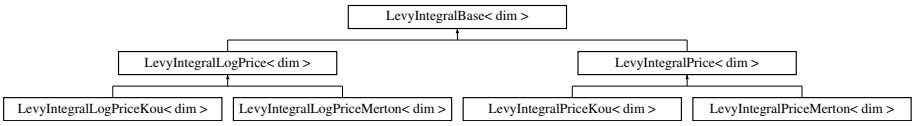

Figura 5.3: Gerarchia delle classi LevyIntegral

5.4.1 LevyIntegralBase<dim>

Questa classe astratta definisce un'interfaccia condivisa e sfruttata da tutte le classi per la quadratura dell'integrale e implementa alcuni metodi di base che possono essere utili alle classi figlie.
I due metodi *core* della classe sono void compute_alpha(), che calcola il valore degli $\hat{\alpha}_i$, e void compute_J(), che calcola il valore della parte integrale da usare nell'*rhs*. Osserviamo inoltre che questa classe base definisce già una funzione che calcola $\hat{\alpha}_i$ in modo generico, ossia con una quadratura composita con nodi di Gauss, e che funziona dunque con qualsiasi modello.

5.4.2 LevyIntegralPrice<dim> e LevyIntegralLogPrice<dim>

La seconda parte dell'integrale, ossia i vettori J_i, è totalmente diversa a seconda che si utilizzi la trasformazione *price* o *log-price*. Per questo motivo abbiamo scritto due classi che ereditano da LevyIntegralBase<dim>. Siccome sono ancora generiche e non sono specializzate su modelli specifici, ereditano dalla classe base il metodo compute_alpha(), che calcola $\hat{\alpha}_i$ per un qualsiasi modello.

La classe LevyIntegralLogPrice<dim> implementa il metodo void compute_J() per la forma *log-price*, che risulta essere più semplice ma allo stesso tempo più lento rispetto al metodo utilizzato dalla trasfomazione *price*. Sia in una che in due dimensioni (e potenzialmente in più), il calcolo di J_i viene svolto ciclando su tutti i nodi della griglia e calcolando in ciascuno il valore dell'integrale. Per fare questo, utilizziamo la classe ausiliaria Solution_Trimmer<dim> e

il metodo get_one_J. Solution_Trimmer<dim> è sostanzialmente un funtore che, inizializzato con la soluzione e la mappatura dei gradi di libertà di deal.ii (cioè, il DoFHandler) agisce diversamente a seconda che il punto stia dentro o fuori dal dominio. Nel primo caso chiama una funzione di deal.ii che individua in che cella si trova il nodo e restituisce il valore della funzione in tale punto, nel secondo caso impone il valore al bordo. La parte computazionalmente costosa è appunto la valutazione in un punto interno al dominio, poiché le funzioni della libreria devono individuare in quale cella si trova quel punto (ricordiamo qui che la funzione incognita va valutata nel punto $y_l + x_i$, dove y_l sono i punti di quadratura e x_i il nodo attuale, quindi per ogni nodo in punti diversi). Una volta ottenuto il valore della soluzione nel punto (sia esso dentro o fuori dal dominio) è possibile effettuare la quadratura dell'integrale. Il calcolo del contributo del nodo i a J_i viene fatto dal metodo get_one_J. Questa struttura è pensata per poter utilizzare un altro tipo di quadratura nelle classi figlie, ridefinendo solo get_one_J. A questo livello, non essendo specificato alcun modello, utilizziamo una quadratura generica (cioè i soliti nodi di Gauss). Per ridurre le operazioni di copia di vettori, compute_J è specializzata sulla dimensione, ma i cambiamenti fra una e due dimensioni sono quasi nulli.

La classe LevyIntegralPrice<dim>, invece, implementa compute_J() specializzando il metodo sul parametro *template*, ottenendo funzioni sostanzialmente diverse. Infatti, come già spiegato nella sezione 3.3, i metodi utilizzati per calcolare la parte integrale J sono fondamentalmente diversi a seconda della dimensione. In una dimensione, per ogni nodo della griglia, occorre integrare scorrendo tutte le celle e calcolandone il contributo. In due dimensioni, il contributo a J_1 nel nodo S_i è calcolato sulla retta monodimensionale parallela all'asse delle ascisse, mentre il contributo a J_2 è calcolato lungo la retta parallela all'asse delle ordinate. Quindi, per ogni nodo, il metodo scorre le facce di tutte le celle della griglia, e, se la faccia sta sulla retta passante per il nodo corrente, ne calcola il contributo. Sebbene questa sia una procedura complicata, richieda una griglia strutturata e richieda che tutti i nodi vengano visitati ma solo alcuni presi in considerazione, essa risulta più rapida rispetto al calcolo con il metodo *log-price*. Infatti, con questo secondo metodo, la soluzione viene valutata nei nodi di quadratura che cadono sulla faccia attuale, quindi le funzioni della libreria conoscono la cella in cui cadono i nodi di quadratura e restituiscono velocemente il valore, senza dover cercare il nodo su tutta la griglia.

5.4.3 Le classi figlie per modelli specifici

Come si nota in figura 5.3, esistono poi delle classi derivate da LevyIntegralPrice <dim> e LevyIntegralLogPrice<dim>. Esse implementano di nuovo i metodi per il calcolo delle parti integrali con dei nodi di quadratura specifici per i vari modelli. In particolare LevyIntegralPriceKou<dim> e LevyIntegralPriceMerton<dim> implementano solo la funzione void compute_alpha() utilizzando rispettivamente nodi di Laguerre e di Hermite, poiché grazie alla trasformazione $y = Se^y$, le densità contro cui integriamo non sono più esponenziali o gaussiane. Le classi LevyIntegralLogPriceKou<dim> e LevyIntegralLogPriceMerton<dim>, invece, oltre a implementare di nuovo void compute_alpha() con i nodi specifici, implementano pure get_one_J sfruttando le potenzialità dei nodi specializzati sui modelli. Nel caso si vogliano aggiungere altri modelli, questo design permette

di creare nuove classi specifiche che ereditano dal secondo livello, permettendo all'utente di modificare solo piccole parti.

5.5 *Factory*

Data la complessità strutturale delle classi Opzione, abbiamo deciso di utilizzare il *design pattern* della *factory* descritta in [3], altrimenti noto come costruttore virtuale, per permettere a un qualsiasi utente di istanziare l'oggetto giusto per il suo problema. Abbiamo quindi creato una classe, Factory che ha costruttore di default, costruttore di copia e operatore di copia privati, perciò non è possibile istanziare la classe, e abbiamo scritto il seguente metodo:

```
static Factory * instance()
        {
                static Factory instance;
                return &instance;
        }
```

Questa funzione permette di istanziare uno e un solo oggetto della classe Factory, che è quindi un *singleton*. Oltre a questo metodo, abbiamo due funzioni std::unique_ptr<OptionBase<dim> > create(...) con il parametro dim specializzato su una e due dimensioni che prendono come argomenti il tipo di opzione, europea o americana, *put* o *call*, il tipo di trasformazione, i modelli dei sottostanti e i vari parametri dell'opzione.

5.6 Come utilizzare la libreria

Sfruttando la *factory* di cui abbiamo appena parlato, utilizzare questa libreria è molto semplice:

- la prima cosa da fare è creare un oggetto di tipo Model, introducendo parametri come il valore del sottostante, la sua volatilità e l'intensità dei salti, nel caso di un modello di Lévy,

  ```
  BlackScholesModel model(95., 0.120381);
  ```

- dopodiché, utilizzando la *factory*, si stabilisce che tipo di opzione istanziare, europea o americana, *put* o *call*, con trasformazione *price* o *log-price*, e si passano i parametri del contratto e di mercato, quali tasso di interesse, scadenza del contratto, prezzo di esercizio, e i parametri di discretizzazione,

  ```
  auto foo=
  Factory::instance()->create(ExerciseType::EU,
  OptionType::Put,
  Transformation::Price,
  model.get_pointer(),
  0.0367, 1., 90., 12, 250);
  ```

- poi, dopo aver settato alcuni parametri e flag a discrezione dell'utente, occorre chiamare la funzione:

```
foo->run();
```

che crea il sistema e lo risolve.

- infine, per stampare il prezzo del titolo finanziario,

```
std::cout<<"The price of the option is "<<foo->
    get_price()<<std::endl;
```

Capitolo 6

Risultati

In questo capitolo riportiamo e commentiamo i risultati ottenuti tramite la nostra libreria. In particolare, analizzeremo nel dettaglio i vari output ottenuti dai programmi test contenuti a titolo esplicativo nel nostro codice. Tutti i risultati seguenti sono stati ottenuti su un computer con un processore *intel i5* quad-core, con 4GB di memoria RAM e la versione 8.1.0 di deal.ii.

6.1 test-1

In questo primo semplice programma, abbiamo calcolato il prezzo di opzioni europee il cui sottostante evolve secondo un classico modello di *Black&Scholes*, ovvero l'equazione da risolvere è la solo PDE, senza parte integrale. I risultati ottenuti sono illustrati nella tabella 6.1. Come possiamo notare i tempi di

Dimensione	Trasformazione	Griglia	Timesteps	Prezzo	Tempo
1	*price*	4096	250	1.42046€	1.36864s
1	*log-price*	4096	250	1.42046€	1.38591s
2	*price*	16384	100	21.5627€	10.1907s
2	*log-price*	16384	100	21.5476€	10.0676s

Tabella 6.1: Prezzi di una *put* 1d e di una *call* 2d.

calcolo sono davvero rapidissimi in tutti i casi. I prezzi dell'opzione monodimensionale sono identici per entrambe le trasformazioni, mentre per il 2d c'è una differenza apprezzabile, e aumentando il numero di elementi nella *mesh* questa differenza scompare. Nelle figure 6.1, 6.3, 6.4 e 6.5 sono riportati, a titolo di esempio i grafici della soluzione e della condizione finale. In figura 6.2 abbiamo plottato la derivata prima e seconda della put europea calcolata tramite la trasformazione *price*. Come possiamo vedere, queste due funzioni sono molto lisce e prive di passaggi bruschi.
Oltre a questi semplici calcoli, abbiamo deciso di testare la convergenza del prezzo al variare della griglia, ottenendo i risultati mostrati nelle tabelle 6.2 e 6.3. Notiamo quindi che in 1d entrambe le trasformazioni, anche con una griglia lasca e pochissimi istanti temporali, danno un risultato molto vicino al prezzo corretto, e poi convergono velocemente al prezzo esatto 9.66. In due dimensioni

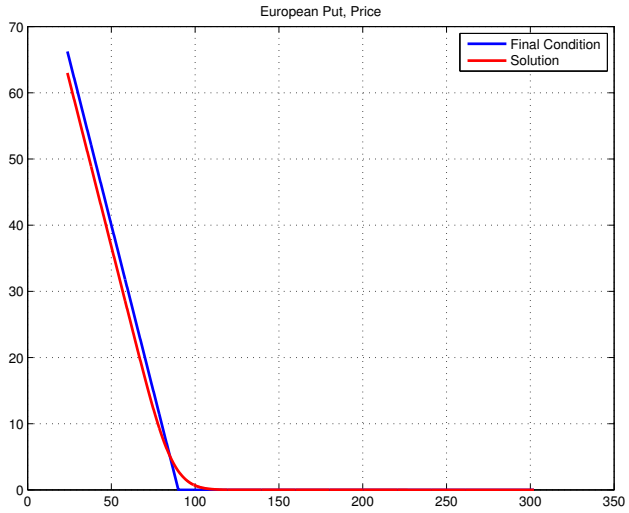

Figura 6.1: *Put* europea, con trasformazione *price*

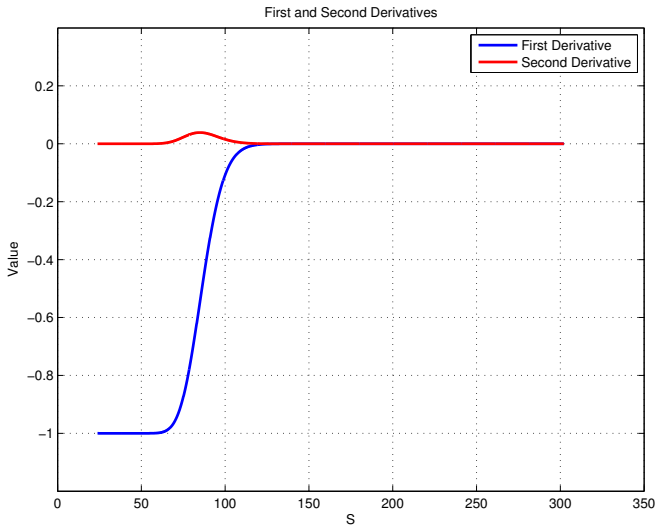

Figura 6.2: Derivata prima e seconda della *put* europea, con trasformazione *price*

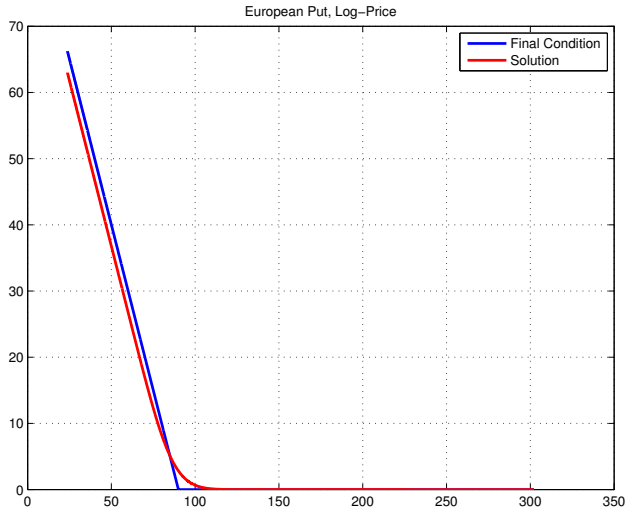

Figura 6.3: *Put* europea, con trasformazione *log-price*

vediamo invece come il nostro *solver* abbia bisogno di parecchi step temporali e una griglia molto fitta per dare il risultato corretto.

Griglia/Timesptes	256/25	512/50	1024/100	2048/250	4196/500
price	9.64979€	9.65598€	9.66063€	9.66336€	9.66431€
log-price	9.64951€	9.65587€	9.66060€	9.66339€	9.66431€

Tabella 6.2: Test di convergenza per una Call 1d.

Griglia/Timesptes	256/25	1024/50	4096/100	16384/250	65536/500
price	5.13714€	2.88788€	2.63722€	2.54018€	2.49968€
log-price	3.42608€	2.79298€	2.54516€	2.51124€	2.49826€

Tabella 6.3: Test di convergenza per una *put* 2d.

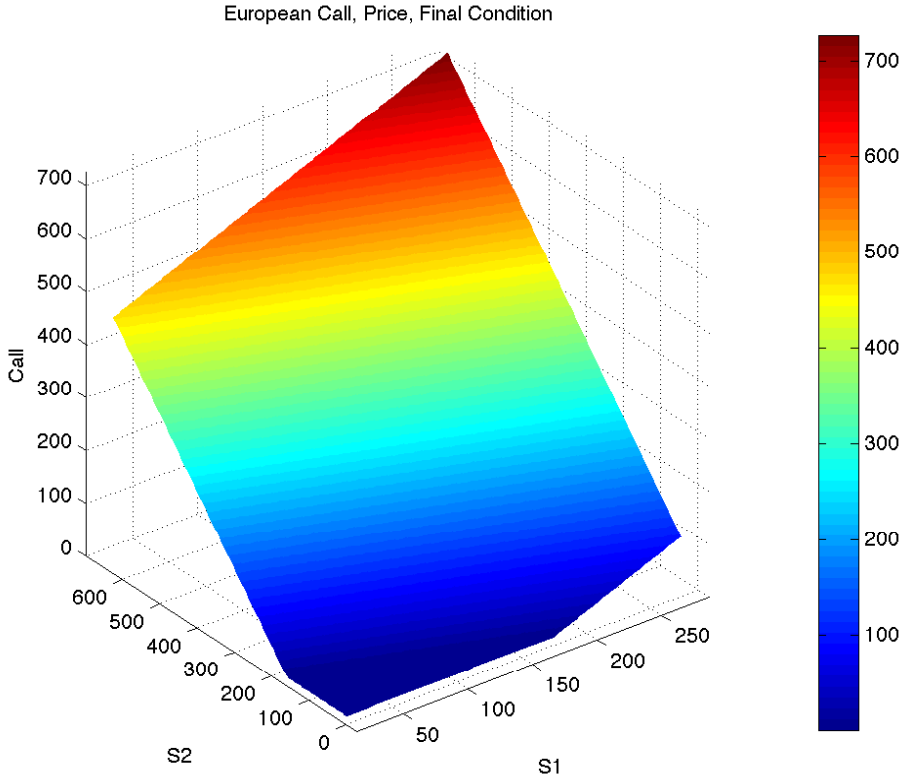

Figura 6.4: *Call* europea, con trasformazione *price*, condizione finale

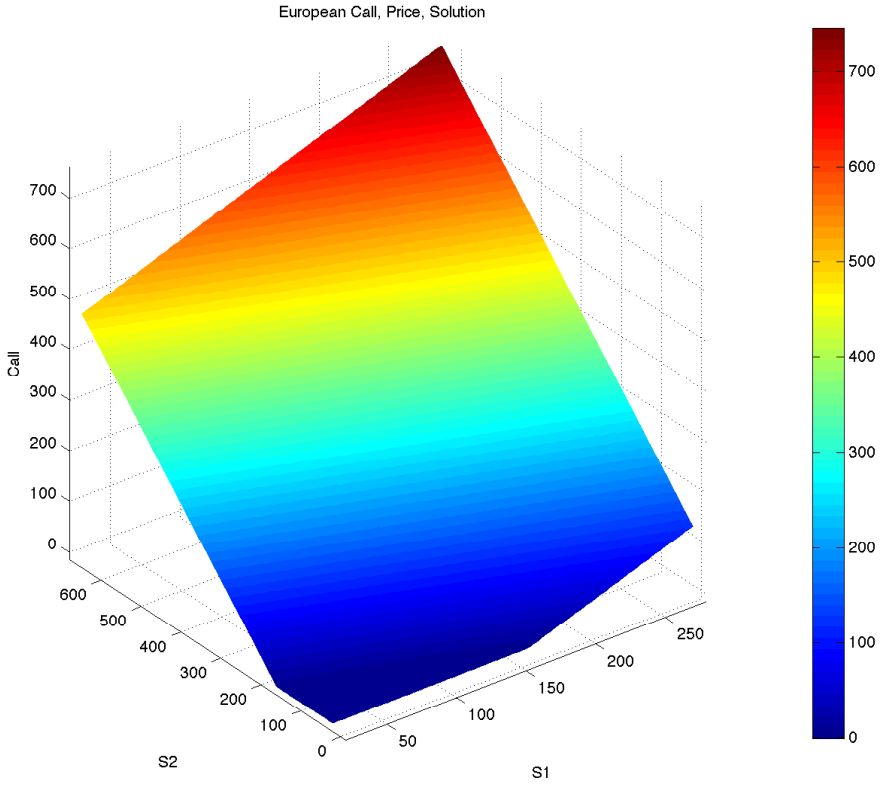

Figura 6.5: *Call* europea, con trasformazione *price*, soluzione

6.2 test-2

In questo secondo programma di esempio calcoliamo i prezzi di opzioni i cui sottostanti evolvono con modelli di Lévy. In particolare, a titolo di esempio abbiamo prezzato delle *call* europee in 1d con un modello di Kou, il cui grafico è riportato in figura 6.6, e in 2d con dei modelli di Merton. I risultati ottenuti sono i riportati nella tabella 6.4.

La prima considerazione da effettuare riguarda i tempi di calcolo: rispetto

Dimensione	Trasformazione	Griglia	Timesteps	Prezzo	Tempo
1	*price*	1024	100	12.4258€	17.3725s
1	*log-price*	1024	100	12.427€	14.3955ss
2	*price*	4096	100	23.9055€	14.6684s
2	*log-price*	4096	100	24.0966€	305.049s

Tabella 6.4: Prezzi di una *call* 1d e 2d.

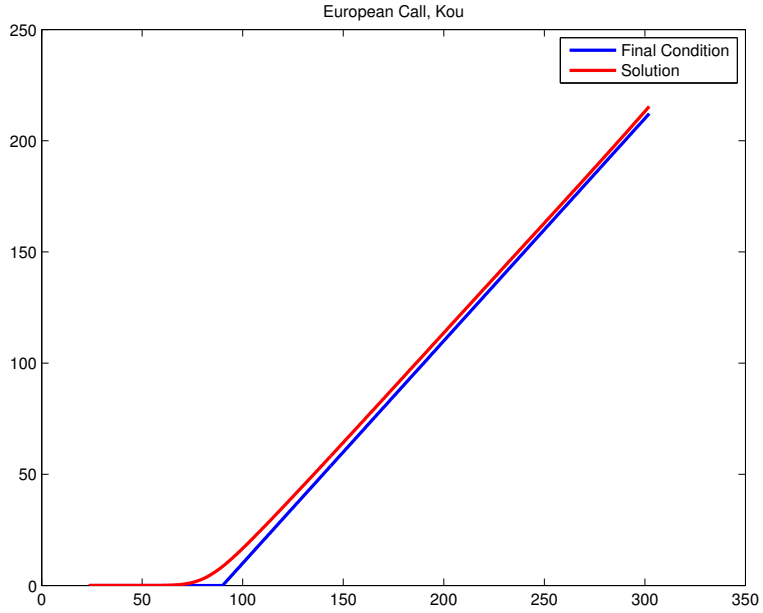

Figura 6.6: *Call* europea, con modello di Kou

al semplice modello di *Black&Scholes*, osserviamo che i tempi impiegati per individuare il prezzo dell'opzioni sono molto maggiori. Nonostante qui le griglie siano più piccole, i tempi di calcolo per la soluzione della PIDE sono più grandi rispetto a quelli della PDE di almeno un ordine di grandezza.
La seconda osservazione riguarda invece il confronto fra le due trasformazioni:

notiamo infatti che in 1d, la trasformazione *log-price* è più veloce rispetto alla trasformazione *price*. Ciò è dovuto al fatto che l'implementazione della parte integrale in *log-price*, pur essendo più pesante rispetto all'altra, poiché occorre ogni volta calcolare dei valori della soluzione in punti non noti a priori della griglia, permette un'immediata parallelizzazione, mentre la trasformazione *price*, in teoria più rapida, non è facilmente parallelizzabile. Quindi, la risoluzione del problema con la trasformazione *price* è sì più veloce da un punto di vista algoritmico, ma in questo particolare caso, con a disposizione 4 processori, la trasformazione *log-price* ha prestazioni di poco migliori.
In due dimensioni, invece, le prestazioni della trasformazione *log-price* non reggono il paragone rispetto a *price*. Questo è dovuto al fatto in due dimensioni la ricerca delle celle in cui si trovano i punti in cui deve essere valutata la soluzione diventa molto più onerosa, trattandosi di una griglia bidimensionale.

Un altro test effettuato in questo programma è il calcolo del prezzo al variare

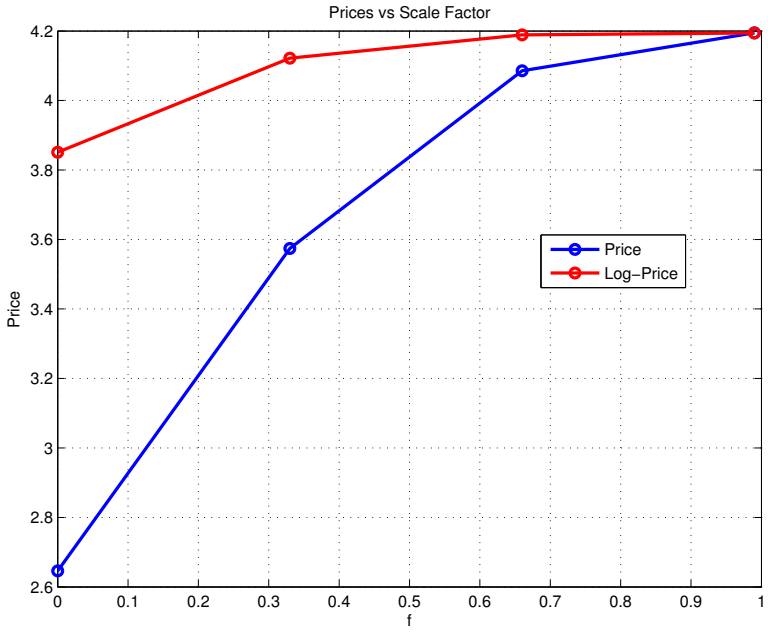

Figura 6.7: Prezzi di una *put* con modello di Kou, al variare di f.

del fattori di scala f, introdotto in (3.2). Testando infatti il comportamento della libreria, ci siamo resi conto che i prezzi delle opzioni erano leggermente inferiori ai prezzi target, specialmente nella trasformazione *price*. Nella figura 6.7 possiamo osservare l'andamento dei prezzi nelle due trasformazioni all'aumentare del fattore di scala. Come accennato in precedenza, un fattore di scala piccolo determina un grosso errore nella valutazione di questo tipo di opzioni nella trasformazione *price*. Ciò è dovuto al fatto che la griglia su cui risolviamo la PDE in questa trasformazione coincide con la griglia su cui calcoliamo l'integrale: se il troncamento è troppo elevato rischiamo di perdere dei contri-

buti importanti della parte integrale. La trasformazione *log-price* invece risente meno di questo problema poiché grazie ai nodi di quadratura di Hermite e di Laguerre il dominio di integrazione non è troncato, ma coincide con tutto l'asse reale, e infatti con questa trasformazione l'errore con f piccolo è contenuto e all'aumentare di f converge velocemente al valore esatto.

Infine, abbiamo studiato come varia il prezzo della trasformazione *log-price* al variare dei nodi di quadratura: il programma permette infatti di adattare la quadratura numerica in modo da scendere al di sotto di errore target. Nella tabella 6.5, mostriamo la variazione di prezzo all'aumentare dei nodi di quadratura.

Come possiamo notare, la differenza di prezzo all'aumentare dei nodi di qua-

# di nodi	4	8	16	32	64	128
Prezzi	12.095€	12.4534€	12.4279€	12.427€	12.4264€	12.4264€
Tempi	4.08526s	6.47795s	9.47712s	14.4233s	20.2579s	28.1988s

Tabella 6.5: Prezzi e tempi di calcolo del prezzo di *call* 1d al variare dei nodi di quadratura.

dratura è davvero irrisoria. Ciò è probabilmente dovuto al fatto che nonostante l'errore sul vettore J possa essere grande, esso viene moltiplicato per la matrice di massa, che generalmente ha numeri molto piccoli. Per questo un errore pur grande su J si manifesta in misura molto attenuata nel prezzo finale. Inoltre, all'aumentare dei nodi di quadratura il tempo di calcolo aumenta considerevolmente. Per questo motivo abbiamo deciso di settare come numero massimo di nodi 32 (ovviamente modificabile con un apposito metodo), in quanto ci sembra l'equilibrio giusto fra precisione e velocità di calcolo.

6.3 test-3

In questo programma abbiamo mostrato come risolvere il problema con l'ostacolo, ovvero come calcolare il prezzo di opzioni americane. Inizialmente, abbiamo calcolato i prezzi di opzioni con modelli di *Black&Scholes* in una e due dimensioni, ottenendo i risultati in 6.6. Il nostro *solver*, che è in grado di lavorare

Dimensione	Trasformazione	Griglia	Timesteps	Prezzo	Tempo
1	*price*	1024	100	1.54933€	0.885832s
1	*log-price*	1024	100	1.54931€	1.08148s
2	*price*	16384	100	4.43963€	1.2148s
2	*log-price*	16384	100	4.41147€	0.990322s

Tabella 6.6: Prezzi di *put* americane 1d e 2d, *Black&Scholes*.

anche in parallelo, sembra comportarsi molto bene, sia per quanto riguarda la velocità di calcolo che la precisione raggiunta: la tolleranza per bloccare il *solver* iterativo è infatti impostata a 10^{-10} e le iterazioni massime sono 1000. Qualora il *solver* non dovesse convergere alla tolleranza in 1000 iterazioni, il programma stampa un *warning*, avvisando l'utente di aver raggiunto il numero massimo di

iterazioni e riportando l'errore attuale. Calcolando i prezzi precedenti con una griglia più fitta il *solver* stampa il *warning*, ma arriva comunque a un errore minore di 10^{-9}, un valore decisamente accettabile.

Oltre al modello di *Black&Scholes*, abbiamo anche utilizzato un modello di Kou, per calcolare il prezzo di un'opzione americana monodimensionale. Con una griglia di 1024 celle e 100 iterazioni temporali otteniamo i seguenti prezzi riportati in 6.7. Come possiamo notare i prezzi sono simili per entrambe le

Trasformazione	Prezzo	Tempo
price	4.42933€	22.9670s
log-price	4.41217€	22.9427s

Tabella 6.7: Prezzi di una *put* americana, Kou.

trasformazioni, come pure i tempi di calcolo. In figura 6.8 possiamo apprezzare come la soluzione stia sempre sopra alla condizione finale, cosa che per esempio non avviene in figura 6.1, che descrive il comportamento di una *put* europea. Nel secondo grafico invece della figura 6.8 abbiamo plottato la derivata prima e la derivata seconda della soluzione: a differenza di quanto accade nella figura 6.2, qui possiamo vedere come esse siano molto "lisce" fino al punto di contatto con l'ostacolo, dove notiamo un cambio un po' brusco di inclinazione. Ciò è dovuto al fatto che da quel punto in poi, nonostante la soluzione tenda a scendere sotto l'ostacolo, il *solver* impone che la soluzione sia pari al *payoff*.

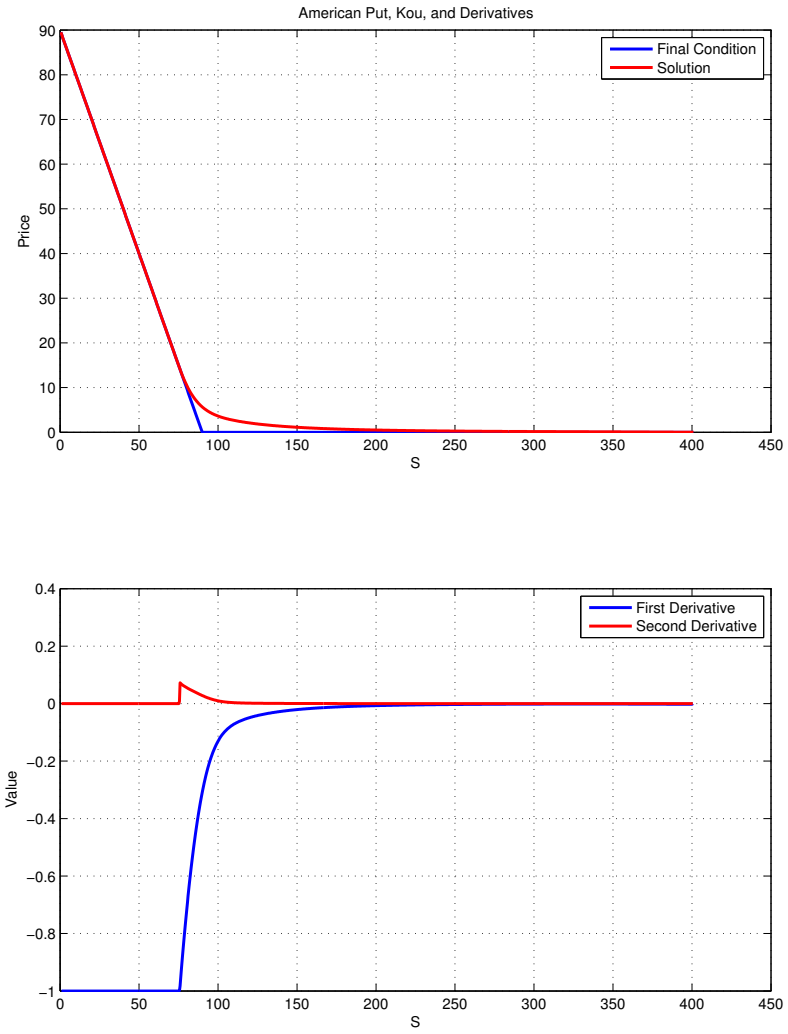

Figura 6.8: *Put* americana, con trasformazione *price*.

6.4 test-4

Il test-4 permette di confrontare le prestazioni e in particolare lo *speed-up* del calcolo dell'integrale in *log-price* sfruttando il calcolo parallelo. Il programma permette di confrontare la velocità del calcolo effettuato con uno e con il numero di processori disponibili sul computer. Oltre a questo, per analizzare nel dettaglio il comportamento del nostro programma, lo abbiamo anche testato con più *threads* rispetto ai processori disponibili. Nelle tabelle 6.8 e 6.9 sono riportati i risultati che riassumono l'andamento dell'accelerazione permessa dal *multi-threading*.

Griglia	Prezzo	Tempo	*speed-up*
1 *thread*:			
256	12.4289€	7.69022s	
512	12.4264€	18.4018s	
1024	12.4264€	48.7283s	
2048	12.4264€	145.728s	
2 *threads*:			
256	12.4289€	4.84786s	1.5863x
512	12.4264€	11.6194s	1.5837x
1024	12.4264€	28.5346s	1.7077x
2048	12.4264€	82.5946s	1.7644x
4 *threads*:			
256	12.4289€	3.63051s	2.11822x
512	12.4264€	7.72159s	2.38316x
1024	12.4264€	18.1312s	2.68754x
2048	12.4264 €	51.4363s	2.83317x
8 *threads*:			
256	12.4289€	4.19548s	1.83298x
512	12.4264€	9.01837s	2.04048x
1024	12.4264€	21.2442s	2.29372x
2048	12.4264€	57.4038s	2.53865x

Tabella 6.8: *Speed test* 1d

Volendo ora applicare la Legge di Amdahl, possiamo calcolare lo *speed-up* teorico massimo possibile e confrontarlo con quello da noi ottenuto. In particolare, la Legge di Amdhal afferma che l'accelerazione massima possibile sfruttando il calcolo parallelo è data dalla seguente equazione:

$$A_{max} = \frac{1}{(1-P) + \frac{P}{S}},$$

in cui A_{max} indica l'accelerazione massima, S il numero di *threads* e P è la frazione di tempo che la parte parallelizzata impiega se eseguita in seriale. Nel nostro caso, come possiamo osservare in figura 4.2, il calcolo dell'integrale effettuato tramite la funzione get_one_J() impiega il 99.59%, quindi dobbiamo porre $P = 0.9959$. L'accelerazione massima teorica è quindi 1.9918x per 2 *cores* e 3.9514x per 4 *cores*. Come possiamo osservare in figura 6.9, non riusciamo a

Griglia	Prezzo	Tempo	*speed-up*
1 *thread*:			
256	19.3305€	19.6544 s	
1024	17.6665€	112.993s	
4096	17.3574€	1029.37s	
16384	17.2674€	12921s	
2 *threads*:			
256	19.3305€	11.1107s	1.7690x
1024	17.6665 €	60.4006s	1.8707x
4096	17.3595€	560.834s	1.8354x
16384	17.2674€	6773.54s	1.9076x
4 *threads*:			
256	19.3305€	7.2931s	2.6949x
1024	17.6665€	39.4015s	2.8677x
4096	17.3595€	348.897s	2.9504x
16384	17.2674€	4229.25s	3.0552x
8 *threads*:			
256	19.3305€	10.4722s	1.87681x
1024	17.6665€	50.4595s	2.23928x
4096	17.3595€	369.883s	2.78296x
16384	17.2674€	42204s	3.06159x

Tabella 6.9: *Speed test* 2d

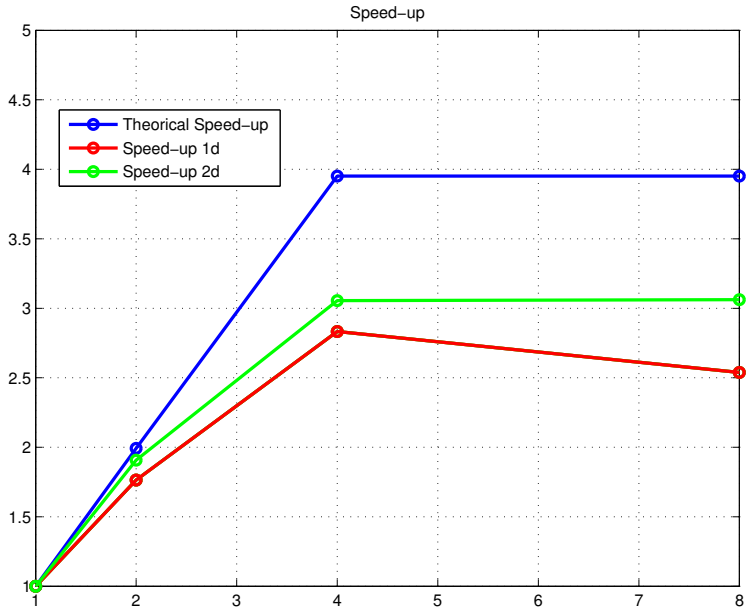

Figura 6.9: Plot degli *speed-up* su un processore quad-core.

raggiungere lo *speed-up* teorico prossimo ai 4x stimato dalla legge di Amdahl, probabilmente perché il computer su cui abbiamo testato il programma non è molto adatto al calcolo parallelo: le prestazioni con due *threads* infatti sono vicine al limite teorico 1.99x, mentre 4 *threads* su 4 *cores* non riescono a sfruttare il 100% delle prestazioni del processore. Riteniamo che il motivo per cui questo accade è che alcune risorse vengano assorbite dal sistema operativo e dalla gestione dei vari processi. Questa conclusione a cui siamo giunti è avvalorata dal fatto che anche con 8 *threads* le prestazioni sono uguali o addirittura peggiorano nell'1d, poiché proprio il computer fatica a gestire molti *threads* computazionalmente onerosi.
In conclusione, non possiamo non notare che i tempi di calcolo sfruttando tutti e quattro i *cores* sono decisamente migliori rispetto al calcolo seriale, permettendo così alla trasformazione *log-price* di competere e addirittura superare da un punto di vista di velocità di calcolo la trasformazione *price* nell'1d.

6.5 test-5

In quest'ultimo test abbiamo valutato le prestazioni della nostra libreria attivando il flag del *mesh refinement*. In particolare, l'abbiamo provato con opzioni monodimensionali in entrambe le trasformazioni, e nel caso bidimensionale per il *log-price*. I risultati ottenuti nel caso 1d sono riportati nella tabella 6.10. Questi

Trasformazione	Celle di partenza	Celle di arrivo	Timesteps	Prezzo	Tempo
price	256	520	100	12.4267€	2.74439s
log-price	256	520	100	12.4281€	4.23711s

Tabella 6.10: Prezzi di una *call* 1d, Kou, con parametri 0.2 di *refinement* e 0.03 di *coarsening*.

risultati sono stati ottenuti con il medesimo modello utilizzato per il caso monodimensionale di 6.4, e come possiamo notare, non solo i prezzi ottenuti sono praticamente gli stessi, ma sono calcolati molto più velocemente, in particolare per la trasformazione *price*.
Oltre alle opzioni europee, abbiamo testato il *mesh refinement* anche sulle opzioni americane, con risultati non pienamente soddisfacenti. Come già detto infatti, i prezzi di queste opzioni viene calcolato mediante un *solver* iterativo, che con l'adattività di griglia fatica a convergere. Abbiamo quindi calcolato i prezzi di una semplice opzione americana con il modello di *Black&Scholes* al variare delle iterazioni massime del *solver*, ottenendo i risultati illustrati nelle tabelle 6.11 e 6.12. I prezzi target sono quelli monodimensionali riportati in 6.6, e come possiamo vedere i prezzi faticano a convergere, anche con 1000 iterazioni massime (ovvero il parametro di default). Ipotizziamo che, a causa del riordinamento dei nodi e aggiunta di altri, la matrice diventi mal condizionata e per questo motivo il *solver* faccia fatica a convergere.

Maxiter	100	1000	10000
Prezzo	1.09713€	1.46016€	1.54845€
Tempo	0.54812s	3.79223s	18.0422s

Tabella 6.11: Prezzi di una *put* americana in *price*, con parametri 0.2 di *refinement* e 0.03 di *coarsening*.

Maxiter	100	1000	10000
Prezzo	1.26063€	1.50763€	1.52639€
Tempo	0.851332s	5.55052es	17.7176s

Tabella 6.12: Prezzi di una *put* americana in *log-price*, con parametri 0.2 di *refinement* e 0.03 di *coarsening*.

Ci siamo infine concentrati sull'adattività di griglia nelle opzioni bidimensionali, con la trasformazione *log-price* (ricordiamo infatti che poiché il *price* richiede una griglia strutturata, non è corretto fare *mesh adapting*). Abbiamo quindi testato il comportamento di una *call* 2d al variare dei coefficienti di adattività, i

cui sottostanti evolvono secondo modelli di Merton, ottenendo i risultati riportati in 6.13. Questi valori sono da confrontare con quelli riportati nella tabella

Ref/Coar	g.d.l di partenza	g.d.l di arrivo	Timesteps	Prezzo	Tempo
0.03/0.15	4225	4780	100	24.1053€	357.969s
0/0.1	4225	3526	100	24.097€	275.006s

Tabella 6.13: Prezzi di una *call* 2d, Merton.

6.4. Come possiamo notare, qui i prezzi convergono al valore giusto, con tempi di calcolo coerenti rispetto alle prestazioni del caso senza raffinamento di griglia: nel primo caso infatti, all'aumentare dei nodi, il programma impiega più tempo a girare, rispetto a quanto indicato in 6.4, mentre nel secondo caso, i nodi della griglia diminuiscono, permettendo all'algoritmo di calcolare il prezzo in modo più rapido. L'accelerazione nel secondo caso tuttavia non è molto evidente, ciò è dovuto sia al fatto che i nodi non diminuiscono così tanto e poi con il *mesh refinement*, il programma ogni 20 iterazioni temporali deve ricostruire il sistema, impiegando un tempo non trascurabile.
Di seguito sono riportate le *mesh* generate nei due casi: nel set con parametri 0.03/0.15 possiamo vedere come la griglia diventi più lasca nella parte in cui la soluzione è nulla e dove la soluzione cresce in modo costante, mentre si infittisce dove la soluzione si alza e vicino ai bordi, dove c'è una condizione al contorno $\mathcal{C}^0$. Nel set di griglie con parametri 0/0.1 notiamo invece come la soluzione diventi più lasca nella zona in cui la soluzione vale 0.

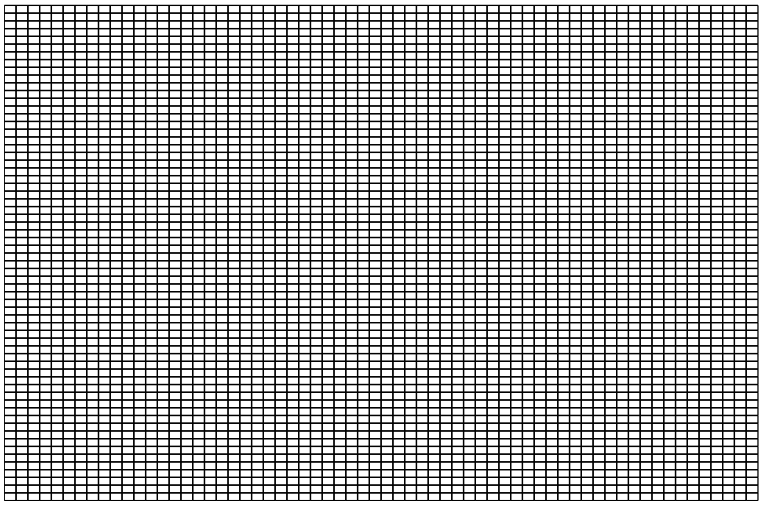

Figura 6.10: Griglia iniziale.

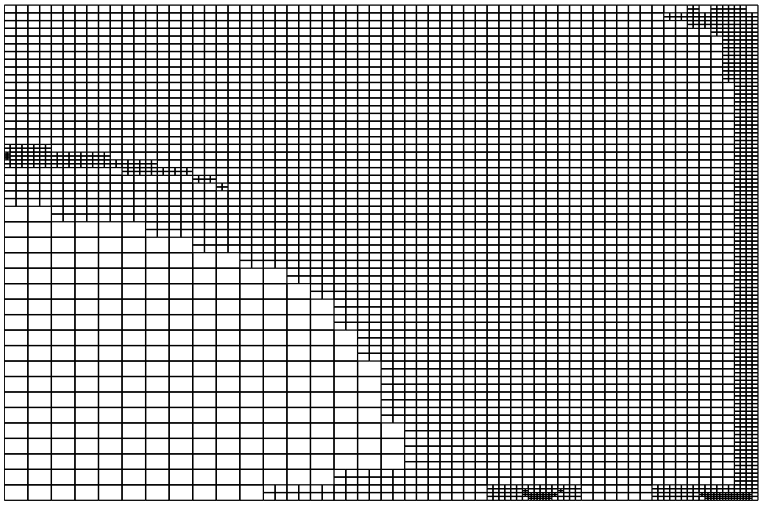

Figura 6.11: Griglia dopo 2 step di raffinamento, con parametri 0.03 di *refinement* e 0.15 di *coarsening*.

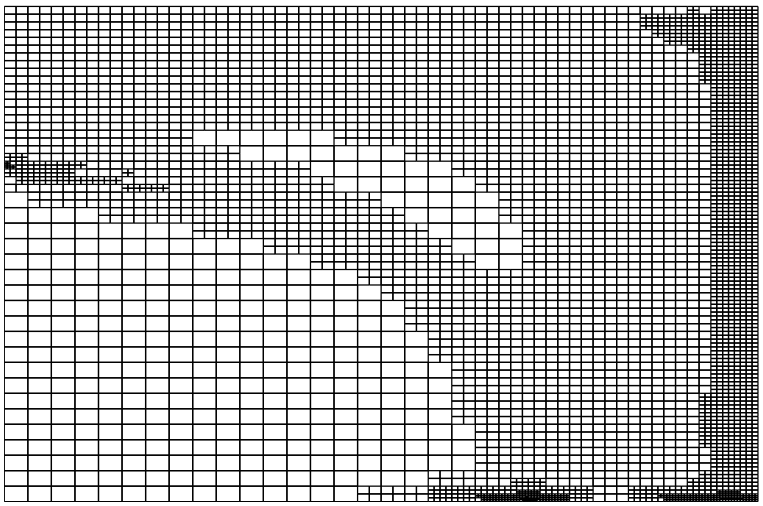

Figura 6.12: Griglia dopo 4 step di raffinamento, con parametri 0.03 di *refinement* e 0.15 di *coarsening*.

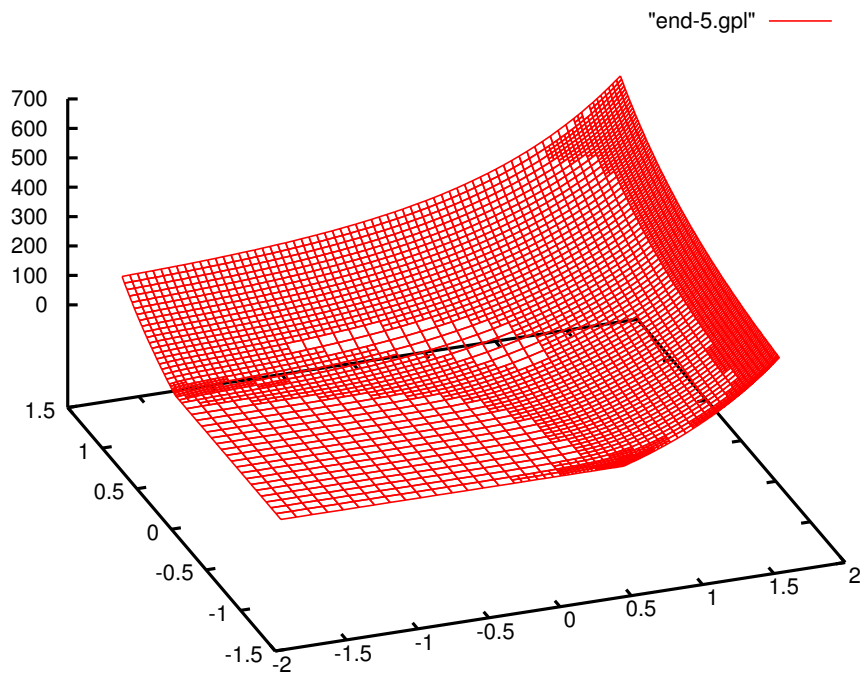

Figura 6.13: Valore della soluzione di una *call* 2d *log-price*, con parametri 0.03 di *refinement* e 0.15 di *coarsening*.

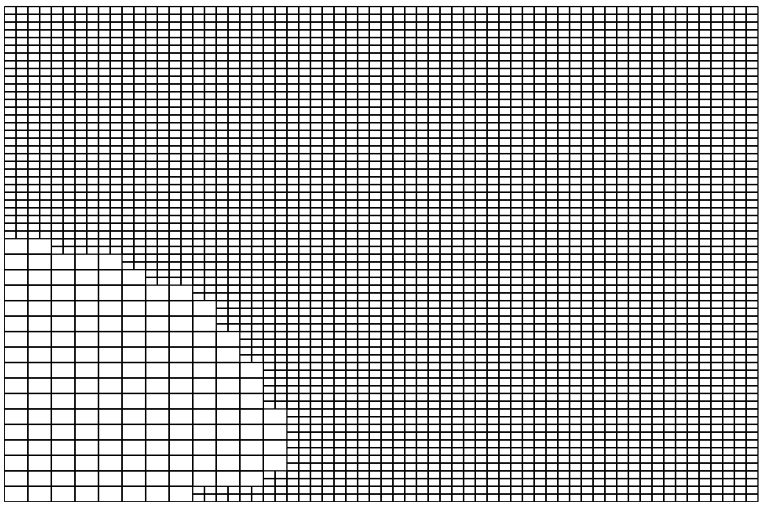

Figura 6.14: Griglia dopo 2 step di raffinamento, con parametri 0 di *refinement* e 0.1 di *coarsening*.

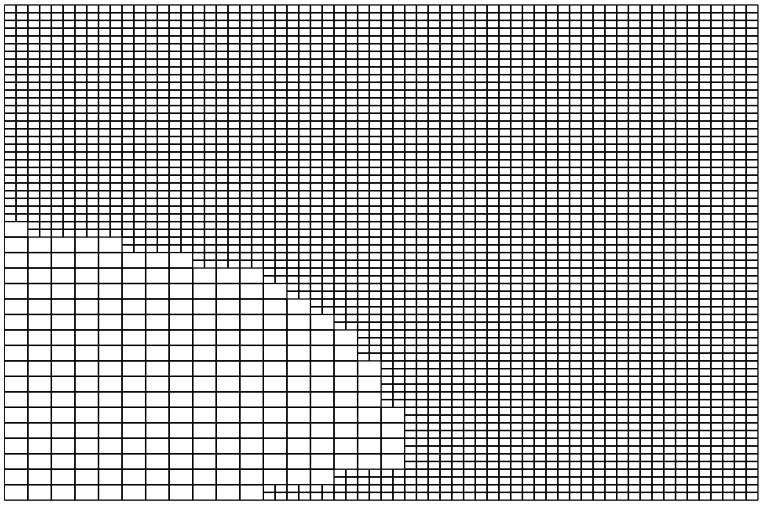

Figura 6.15: Griglia dopo 4 step di raffinamento, con parametri 0 di *refinement* e 0.1 di *coarsening*.

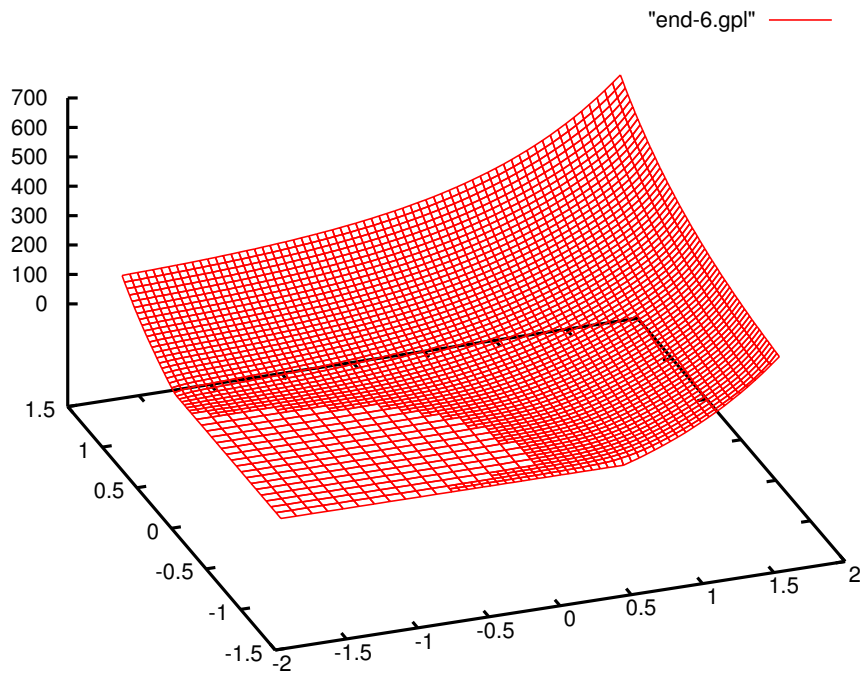

Figura 6.16: Valore della soluzione di una *call* 2d *log-price*, con parametri 0 di *refinement* e 0.1 di *coarsening*.

Capitolo 7

Estensioni

Il programma che abbiamo scritto si presta molto a possibili estensioni, su molti livelli. In particolare, a partire dalle classi base OptionBasePrice<dim> e OptionBaseLogPrice<dim> è possibile scrivere dei *solver* che prezzino tutte le opzioni barriera, ovvero opzioni caratterizzate da condizioni al bordo di *Dirichlet* nulle in alcune parti o su tutto il bordo dominio, che riflettono il fatto che se il sottostante tocca tali barriere l'opzione ha valore nullo.
È inoltre possibile estendere la libreria in modo che prezzi opzioni con altri modelli, per esempio ad attività infinita, caratterizzati cioè da misure di Lévy non limitate (come, ad esempio, il *Normal Inverse Gaussian* e il *Variance Gamma*). Per fare ciò occorre modificare le classi integrale, in modo che calcolino il parametro λ, che per questi modelli è incognito e calcolare all'interno della classe opzione la diffusione che approssima i piccoli salti (che in questo caso, sono infiniti).
Un'altra estensione possibile, ma abbastanza delicata, è estendere il programma al calcolo di opzioni con tre sottostanti, ovvero risolvere un PIDE 3d. Teoricamente l'equazione sarebbe formalmente la stessa, con l'aggiunta di un terzo integrale su $\mathbb{R}$. Occorrerebbe tuttavia prestare attenzione a dove integrare i vari integrali: in 2d, infatti, nella trasformazione *price* abbiamo calcolato gli integrali sui lati dei quadrilateri che costituiscono la *mesh*, in 3d si dovrebbe invece integrare sui lati delle facce dei parallelepipedi. Teoricamente, così come deal.ii offre la possibilità di lavorare sulle facce di celle bidimensionali, permette anche di lavorare su facce e spigoli nel caso tridimensionale.
Si potrebbe provare poi a utilizzare le funzioni offerte dalla libreria deal.ii per la soluzione del problema con memoria distribuita. Per quanto riguarda la parte PDE si può sfruttare quanto implementato nella libreria, mentre per il calcolo dell'integrale in *price* occorrerebbe spedire tutta la soluzione a tutti i processi (perché il termine è non locale), in *log-price* invece tutti i processi devono conoscere tutta la griglia, per poter valutare la funzione nei punti $y + x$ (3.5).

Proprio perché molto semplice, abbiamo creato due piccole estensioni a titolo illustrativo.

doubling_extension
Un estensione che cambia il modo di stimare l'errore per il *mesh refinement*. Ereditando da EuropeanOptionLogPrice, reimplementa i metodi solve e refine_grid,

e definisce due nuovi metodi: solve_one_step e estimate_doubling. Per stimare l'errore, si procede nel modo seguente. La soluzione e la griglia attuale vengono messe da parte, e si raffina ogni cella della griglia, ottenendo una griglia fine. Su questa si trasferisce la soluzione tramite interpolazione e si risolve il problema, ottenendo una nuova soluzione sulla griglia fine. Questa soluzione ottenuta sulla griglia più fine viene poi interpolata sulla griglia originale, ed in ogni cella si calcola, per ogni grado di liberta della cella, l'errore scarto quadratico ra soluzione sulla griglia fine e quella *coarse*. L'errore nella cella è dato dalla somma degli errori dei suoi gradi di libertà. In seguito si procede a raffinare le celle con un errore più grande e deraffinare quelle con un errore più basso.

barrier
Questa estensione, permette di calcolare il prezzo di una opzione barriera *call up&out*. Questo tipo di opzioni sono caratterizzate da un *payoff* nella forma seguente:

$$C(S(t),t) = max(S_T - K, 0)\mathcal{I}_{\{S_t<H, t\in[0,T]\}},$$

ovvero queste opzioni pagano il *payoff* a scadenza se e solo se il sottostante non ha mai superato la barriera H in $[0,T]$. Per il *pricing* di queste opzioni occorre in primo luogo troncare il dominio in corrispondenza della barriera H e imporre condizioni al bordo nulle sui lati della *mesh* in cui è presente la barriera. Come possiamo notare in figura 7.1, il problema ha in questo caso condizioni al contorno nulle ovunque: su due lati infatti, occorre imporre la condizione della *call* (che è nulla), sugli altri due lati invece si impongono condizioni al bordo nulle per la presenza delle barriere.

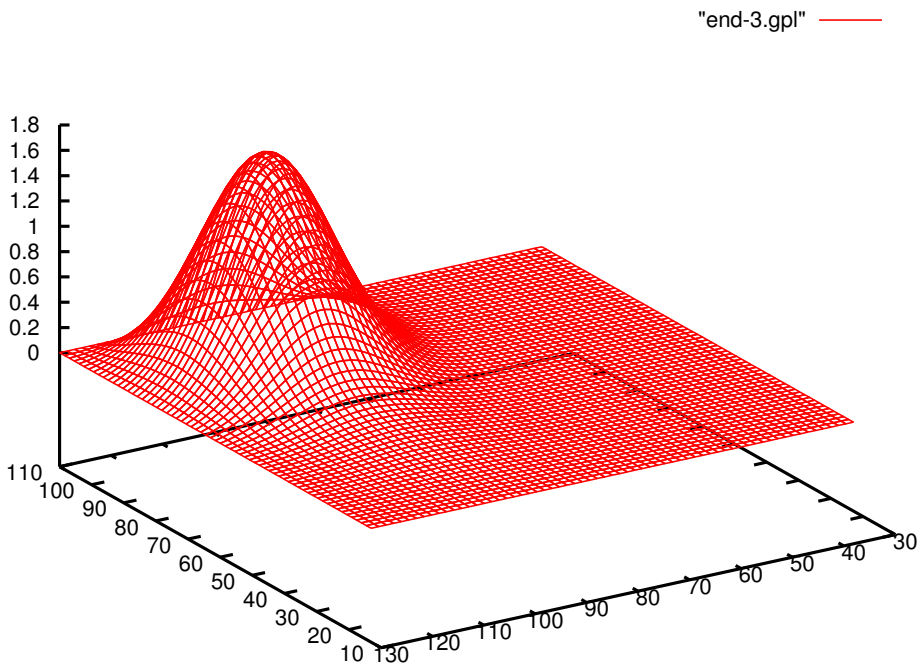

Figura 7.1: Valore della soluzione di una *call up&out*, con barriere $H_1 = 110$, $H_2 = 130$.

Capitolo 8

Conclusioni

Siamo dunque giunti al termine di questo elaborato, ed è ora di trarre delle conclusioni.

Il primo punto su cui vorremmo soffermarci è il confronto fra le trasformazioni *price* e *log-price*: come abbiamo potuto vedere nel Capitolo 6, analizzando i risultati, la trasformazione *price* in 2d si è dimostrata sorprendentemente veloce nella risoluzione del problema, mentre nel caso monodimensionale occorre fare delle considerazioni più delicate. Nonostante infatti il calcolo dell'integrale con *price* sia algoritmicamente più veloce rispetto alla trasformazione *log-price*, quest'ultima, complici la facile parallelizzazione e la veloce convergenza dell'integrale garantita dai nodi di Hermite e di Laguerre, risulta più veloce su un computer di media potenza, cioè con a disposizione 4 *cores*.

Un altro punto su cui vorremmo spendere qualche parola riguarda il *solver* iterativo che abbiamo scritto per il *pricing* di opzioni americane. Anche in questo caso abbiamo ottenuto dei risultati contrastanti: per quanto riguarda il caso *Black&Scholes* senza *mesh refinement* infatti le prestazioni sono ottime, anche migliori del *solver* UMFPACK di deal.ii come dimostra il confronto fra i risultati nelle tabelle 6.1 e 6.6, mentre quando introduciamo il *mesh refinement* le prestazioni peggiorano molto, tanto che nel *log-price* il *solver* fatica addirittura a convergere al risultato corretto. Riteniamo che questi comportamenti contrastanti siano legate al condizionamento del problema: il nostro *solver* infatti è molto semplice e non fa alcun test sulla matrice di sistema, e questo si traduce in prestazioni ottime quando la matrice è ben condizionata e prestazioni non buone quando è mal condizionata. Il *solver* di deal.ii invece riteniamo che esegua dei test sulla matrice, impiegando anche del tempo aggiuntivo per risolvere il sistema, e applichi in ogni occasione il metodo migliore[1]. Probabilmente, applicando dei condizionatori adatti, le prestazioni del nostro *solver* iterativo potrebbero beneficiarne.

Dedichiamo poi qualche riga al confronto fra metodi alle differenze finite e metodi agli elementi finiti. Come già accennato in precedenza, non possiamo non annoverare fra i vantaggi degli elementi finiti il fatto che la soluzione è calcolata

[1]Nella descrizione di UMFPACK infatti è scritto che questo *tool* è un set di *routine* per risolvere sistemi lineari asimmetrici, quindi non è semplicemente un *solver*.

sull'intero dominio, e non solo su un numero finito di punti come nel caso delle differenze finite. Ciò permette quindi di ottenere una soluzione più "pregiata", utile anche nel caso in cui occorra calcolare le derivate della soluzione. A tal proposito è noto che gli elementi finiti riescano meglio rispetto alle differenze finite a descrivere cambi relativamente bruschi di inclinazione. L'adattamento di griglia infatti può essere adottato per il calcolo della soluzione anche con condizioni finali continue ma non derivabili, ottenendo così un risultato più preciso non solo nel prezzo, ma anche nelle misure di sensitività del derivato. Le cosiddette Greche, infatti, cioè i valori delle derivate prima e seconda della soluzione, possono essere calcolate con più precisione rispetto al caso delle differenze finite. Ovviamente il lato negativo degli elementi finiti è la difficoltà sia nel ricavare la formulazione debole, sia nell'implementare correttamente i vari algoritmi da un punto di vista di programmazione. Infatti, mentre con le differenze finite anche un problema in 2d è facilmente implementabile, con gli elementi finiti è molto complicato risolvere problemi multi dimensionali senza l'ausilio di una libreria esterna che si occupi di gestire gli elementi. La piccola libreria da noi scritta, con l'ausilio di deal.ii, nasconde parte di queste problematiche e permette di sviluppare codici agli elementi finiti in modo molto rapido, anche nel caso bidimensionale.

Infine ci soffermiamo a sottolineare la velocità dei nostri codici rispetto a una semplice implementazione in Matlab. Nel caso integro-differenziale monodimensionale infatti, i nostri prezzi vengono calcolati in un terzo del tempo. Nel caso 2d, invece, abbiamo potuto confrontare i tempi solo con un caso *Black&Scholes* alle differenze finite, e anche in questo caso il nostro codice è due volte più veloce.

Bibliografia

[1] Quarteroni Alfio, Sacco Riccardo, and Saleri Fausto. *Numerical Mathematics*. Springer, 2007.

[2] Tomas Bjork. *Arbitrage Theory in Continuous Time*. Oxford Finance, 2009.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[4] Jiang Tao, Liu Xin, and Yu Zhengzhou. Finite element algorithms for pricing 2-d basket options. In *Information Science and Engineering (ICISE), 2009 1st International Conference on*, pages 4881–4886. IEEE, 2009.

[5] Jinghui Zhou. *Multi-Asset Option Pricing with Levy Process*. PhD thesis, South China University of Technology, 2009.