Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea Magistrale in Matematica

Calcolo parallelo per modelli stocastici con default

Tesi di Laurea in Equazioni Differenziali Stocastiche

Relatore:
Chiar.mo Prof.
Andrea Pascucci

Presentata da:
Graziana Colonna

Sessione III
Anno Accademico 2015-2016

Introduzione

Un aspetto importante nella valutazione del prezzo di un'opzione, strumento finanziario che si è diffuso negli ultimi decenni, è la gestione del rischio. Il rischio di credito può essere definito come l'eventualità che il debitore non onori gli obblighi finanziari assunti, causando una perdita da parte della controparte creditrice. Questa eventualità rende la valutazione del prezzo di un'opzione ancora più complessa. Per questo motivo sono stati introdotti i modelli con default (insolvenza), oggetto di studio di questa tesi.
Strettamente connesso alla gestione del rischio e a strumenti derivati tra cui le opzioni, è il Credit Default Swap (CDS). Esso è uno swap, ovvero uno dei più moderni strumenti di copertura dei rischi utilizzato prevalentemente dalle banche, dalle imprese e anche dagli enti pubblici, che ha la funzione di trasferire il rischio di credito.
Per descrivere lo schema di un CDS, supponiamo che ci sia un investitore A che abbia un credito nei confronti di una controparte B. Con il fine di proteggersi dal rischio che la parte B fallisca e che il credito perda di valore, l'investitore A si rivolge a una terza parte C, disposta ad accollarsi tale rischio. Da un lato, la parte A si impegna a versare a C un importo periodico, il cui ammontare è il principale oggetto dell'accordo contrattuale; dall'altro C si impegna a rimborsare alla parte A il valore nominale del titolo di credito, nel caso in cui il debitore B diventi insolvente.
Limitandoci ad opzioni di tipo europeo, calcolarne il prezzo, molto spesso, si riduce al dover calcolare il valore atteso di una variabile aleatoria. Uno dei metodi numerici più diffusi per il calcolo di un valore atteso, è il metodo

Monte Carlo, che ha molti vantaggi ma anche uno svantaggio non trascurabile: i tempi di performance. Se volessimo calibrare i parametri di un modello stocastico con default attraverso i CDS, il metodo Monte Carlo necessiterebbe di milioni e milioni di calcoli e, conseguentemente, di tempi di esecuzione estremamente lunghi. Questa è la ragione che ci ha indotti a provare un tipo di calcolo in parallelo, supportato dall'architettura hardware CUDA, nella speranza che i tempi di performance subissero un netto miglioramento. La tesi è suddivisa in cinque capitoli:

Capitolo 1: presentiamo la definizione rigorosa di tempo di default, ed enunciamo dei risultati matematici che ci permetteranno costruire un modello che tenga conto della gestione del rischio di credito. In particolare, costruiamo, a partire dal tempo di default τ, una filtrazione che contenga anche le informazioni relative alla possibilità di default ed enunciamo delle proprietà con cui poter gestire più facilmente, matematicamente parlando, questa filtrazione.

Capitolo 2: qui presentiamo il nostro modello stocastico, a partire dalla definizione di un particolare tipo di tempo di default. Dimostriamo che le proprietà enunciate nel primo capitolo valgono per il tempo di default così definito e le applichiamo per calcolare il prezzo di un'opzione di tipo europeo. In particolare, presentiamo due modelli, JDCEV e Vasicek che verranno utilizzati nel resto della tesi per valutare il prezzo di un'opzione di tipo europeo.

Capitolo 3: in questo capitolo viene presentato il metodo Monte Carlo, uno dei metodi più conosciuti per il calcolo del valore atteso di una variabile aleatoria. Alla base del Monte Carlo, ci sono i generatori di numeri casuali. Descriviamo dettagliatamente i generatori di numeri casuali uniformemente distribuiti nell'intervallo $[0, 1]$ e di numeri casuali con distribuzione normale di media 0 e varianza 1. Riportiamo l'algoritmo Monte Carlo utilizzato per calcolare il prezzo dell'opzione europea, la cui formula è stata presentata nel

secondo capitolo.

Capitolo 4: introduciamo un tipo di calcolo diverso da quello sequenziale a cui siamo solitamente abituati: il calcolo in parallelo. Descriviamo cosa è, come funziona e presentiamo CUDA, un particolare tipo di architettura hardware in grado di supportare il calcolo in parallelo. Attraverso un semplice esempio (generazione di N sequenze di numeri con distribuzione normale), mostriamo le differenze di performance tra il calcolo sequenziale e quello parallelo. Presentiamo, infine, i primi risultati dell'algoritmo Monte Carlo per prezzare uno Zero Coupon Bond.

Capitolo 5: Introduciamo il problema della calibrazione di un modello matematico e presentiamo l'algoritmo Nelder-Mead per i problemi di ottimizzazione. Definiamo di un particolare tipo di derivato, Credit Default Swap (CDS), e presentiamo delle formule esplicite con cui calcolare le survival probabilities. Calibriamo il nostro modello sulle survival probabilities dei CDS, prima considerando i coefficienti costanti in tutto l'intervallo temporale, poi considerandoli costanti a tratti, e mostriamo i risultati ottenuti dopo le calibrazioni.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Rischio di credito

Il rischio di credito può essere definito come l'eventualità che una delle parti di un contratto non onori gli obblighi di natura finanziaria assunti, causando una perdita per la controparte creditrice. Gestire il rischio di credito è un aspetto molto importante in ambito finanziario. Per poterlo fare, è necessario introdurre il concetto di tempo di default. Esso, in poche parole, rappresenta l'istante in cui avviene l'insolvenza da parte del debitore. Per descrivere rigorosamente il modello con cui vogliamo prezzare un'opzione, tenendo conto del rischio, sono necessarie alcune premesse matematiche, che facciamo di seguito.

1.1 Risultati preliminari

Prima di tutto, facciamo alcuni richiami su concetti di calcolo delle probabilità, che ci serviranno per costruire il nostro modello.

Definizione 1.1. Un processo stocastico su $\mathbb{R}$ è una collezione di variabili aleatorie $(X_t)_{t\geq 0}$ su uno spazio di probabilità $(\Omega, \mathscr{G}, \mathbb{P})$ tale che la mappa $X : \mathbb{R}^+ \times \Omega \to \mathbb{R}$

$$X(t, \omega) := X_t(\omega)$$

sia misurabile rispetto alla $\sigma-$algebra prodotto $B \otimes \mathscr{G}$.

Definizione 1.2. Una filtrazione su uno spazio di probabilità $(\Omega, \mathscr{G}, \mathbb{P})$ è una famiglia $\mathbf{G} = (\mathscr{G}_t)_{t\geq 0}$ di sotto $\sigma-$algebre di $\mathscr{G}$ tale che $\mathscr{G}_t \subseteq \mathscr{G}_s$ per ogni $t \leq s$. $(\Omega, \mathscr{G}, \mathbb{P}, (\mathscr{G}_t)_{t\geq 0})$ è uno spazio di probabilità con filtrazione. Intuitivamente si può pensare ad una $\sigma-$algebra $\mathscr{G}_t$ come l'informazione nota al tempo t.

Definizione 1.3. Un processo $(X_t)_{t\geq 0}$ si dice adattato alla filtrazione $\mathbf{G}$, se X_t è $\mathscr{G}_t-$misurabile, per ogni $t \geq 0$.

Il nostro obiettivo è quello di costruire una filtrazione $\mathbf{G}$, che tenga anche conto della possibilità di default.

Definizione 1.4. In uno spazio con filtrazione $(\Omega, \mathscr{G}, \mathbb{P}, (\mathscr{G}_t)_{t\geq 0})$, una variabile aleatoria

$$\phi : \Omega \to \mathbb{R}^+ \cup +\infty$$

è un tempo d'arresto rispetto alla filtrazione $\mathbf{G}$ se

$$\{\phi \leq t\} \in \mathscr{G}_t, \quad \forall t \in \mathbb{R}^+.$$

Sia $\tau : \Omega \to \mathbb{R}$ una variabile aleatoria non negativa sullo spazio di probabilità $(\Omega, \mathscr{G}, \mathbb{P})$ tale che:

- $\mathbb{P}(\tau = 0) = 0$;
- $\mathbb{P}(\tau > t) > 0$ per ogni $t \in \mathbb{R}^+$.

Definizione 1.5. Si definisce processo di salto H associato al tempo τ il processo determinato dalla seguente equazione:

$$H_t = \mathbb{1}_{\{\tau \leq t\}} \quad \forall t \in \mathbb{R}. \tag{1.1}$$

Denotiamo con $\mathbf{H}$ la filtrazione $(\mathscr{H}_t)_{t\geq 0}$ generata dal processo H, ovvero $(\mathscr{H}_t)$ è la $\sigma-$algebra così definita:

$$\begin{aligned} \mathscr{H}_t &= \sigma(H_u | u \leq t); \\ \mathscr{H}_\infty &= \sigma(H_u | u \in \mathbb{R}^+). \end{aligned}$$

Denotiamo invece con $\mathbf{G}$ la filtrazione $(\mathscr{G}_t = \mathscr{F}_t \vee \mathscr{H}_t)_{t\geq 0}$, dove $\mathscr{F}_t$ è una qualunque filtrazione definita sullo spazio di probabilità $(\Omega, \mathscr{G}, \mathbb{P})$. Per costruzione, τ è banalmente un tempo d'arresto rispetto alla filtrazione $\mathbf{H}$. Osserviamo che il processo H è $\mathbf{G}$-adattato, dato che, per definizione, $\mathscr{H}_t \subseteq \mathscr{G}_t$ per ogni $t \in \mathbb{R}^+$, cioè τ è un tempo d'arresto rispetto alla filtrazione $\mathbf{G}$.

Definizione 1.6. Sia X una variabile aleatoria sommabile su uno spazio di probabilità $(\Omega, \mathscr{G}, \mathbb{P})$ e sia $\mathscr{D}$ una sotto σ-algebra di $\mathscr{G}$. Si definisce *attesa condizionata* $E[X|\mathscr{D}]$ di X rispetto a $\mathscr{D}$ la variabile aleatoria Y tale che:

- Y è integrabile e $\mathscr{D}$-misurabile;
- $\int_A Y dP = \int_A X dP$ per ogni $A \in \mathscr{D}$.

Denotiamo con $\mathbb{P}(B|\mathscr{D})$ l'attesa condizionata $E[\mathbb{1}_B|\mathscr{D}]$, per ogni $B \in \mathscr{G}$.

Definizione 1.7. Sia G_t il processo così definito ($\mathbf{F}$-survival process):

$$G_t = \mathbb{P}(\tau > t|\mathscr{F}_t) \quad \forall t \in \mathbb{R}^+.$$

Banalmente, dalla definizione di attesa condizionata, G_t è $\mathscr{F}_t$-misurabile.

Definizione 1.8. Supponiamo che G_t sia strettamente maggiore di zero per ogni $t \in \mathbb{R}^+$. Allora è possibile definire il seguente processo ($\mathbf{F}$-hazard process):

$$\Gamma_t = -\ln(G_t) \quad \forall t \in \mathbb{R}^+.$$

Lemma 1.1.1. *Sia Y una variabile aleatoria sullo spazio di probabilità $(\Omega, \mathscr{G}, \mathbb{P})$. Allora per ogni $t \in \mathbb{R}^+$ risulta:*

$$E[\mathbb{1}_{\{\tau>t\}}Y|\mathscr{G}_t] = \mathbb{P}(\tau > t|\mathscr{G}_t)\frac{E[\mathbb{1}_{\{\tau>t\}}Y|\mathscr{F}_t]}{G_t}.$$

Dimostrazione. La prima proprietà della definizione di attesa condizionata è banalmente verificata.

Considerando le proprietà dell'attesa condizionata e la sua definizione, l'equazione da verificare è equivalente alla seguente:

$$E[\mathbb{1}_{\{\tau>t\}}Y\mathbb{P}(\tau>t|\mathscr{F}_t)|\mathscr{G}_t] = E[\mathbb{1}_{\{\tau>t\}}E[\mathbb{1}_{\{\tau>t\}}Y|\mathscr{F}_t]|\mathscr{G}_t]$$

$$\Updownarrow$$

$$\int_A \mathbb{1}_{\{\tau>t\}}Y\mathbb{P}(\tau>t|\mathscr{F}_t)d\mathbb{P} = \int_A \mathbb{1}_{\{\tau>t\}}E[\mathbb{1}_{\{\tau>t\}}Y|\mathscr{F}_t]d\mathbb{P}$$

per ogni $A \in \mathscr{G}_t$.

Osserviamo che per ogni $A \in \mathscr{G}_t$ esiste un elemento $B \in \mathscr{F}_t$ tale che $A \cap \{\tau > t\} = B \cap \{\tau > t\}$. Infatti, sapendo che $\mathscr{F}_t \subseteq \mathscr{G}_t$, se $A \in \mathscr{F}_t$ basta prendere $B = A$; altrimenti, se $A \notin \mathscr{F}_t$, vuol dire che $A \in \mathscr{H}_t$ per qualche $t \geq 0$. Basta quindi dimostrarlo per $A = \{\tau \leq u\}$. Anche in questo caso la tesi è verificata: è sufficiente prendere $B = \emptyset$.

Ritornando all'ultima uguaglianza che vogliamo dimostrare e applicando alcune proprietà dell'attesa condizionata, abbiamo che:

$$\begin{aligned}
\int_A \mathbb{1}_{\{\tau>t\}}Y\mathbb{P}(\tau>t|\mathscr{F}_t)d\mathbb{P} &= \int_{A\cap\{\tau>t\}} Y\mathbb{P}(\tau>t|\mathscr{F}_t)]d\mathbb{P} = \\
&= \int_{B\cap\{\tau>t\}} Y\mathbb{P}(\tau>t|\mathscr{F}_t)]d\mathbb{P} = \\
&= \int_B \mathbb{1}_{\{\tau>t\}}Y\mathbb{P}(\tau>t|\mathscr{F}_t)d\mathbb{P} = \\
&= \int_B E[\mathbb{1}_{\{\tau>t\}}Y|\mathscr{F}_t]E[\mathbb{1}_{\{\tau>t\}}|\mathscr{F}_t]d\mathbb{P} = \\
&= \int_B E[\mathbb{1}_{\{\tau>t\}}E[\mathbb{1}_{\{\tau>t\}}Y|\mathscr{F}_t]|\mathscr{F}_t]d\mathbb{P} = \\
&= \int_B \mathbb{1}_{\{\tau>t\}}E[\mathbb{1}_{\{\tau>t\}}Y|\mathscr{F}_t]d\mathbb{P} = \\
&= \int_{B\cap\{\tau>t\}} E[\mathbb{1}_{\{\tau>t\}}Y|\mathscr{F}_t]d\mathbb{P} = \\
&= \int_{A\cap\{\tau>t\}} E[\mathbb{1}_{\{\tau>t\}}Y|\mathscr{F}_t]d\mathbb{P} = \\
&= \int_A \mathbb{1}_{\{\tau>t\}}E[\mathbb{1}_{\{\tau>t\}}Y|\mathscr{F}_t]d\mathbb{P}.
\end{aligned}$$

□

Corollario 1.1.2. *Sia $\mathscr{H}_t \subseteq \mathscr{G}_t$, allora:*

$$E[\mathbb{1}_{\{\tau>t\}}Y|\mathscr{G}_t] = \mathbb{1}_{\{\tau>t\}}\frac{E[\mathbb{1}_{\{\tau>t\}}Y|\mathscr{F}_t]}{G_t} = \mathbb{1}_{\{\tau>t\}}E[\mathbb{1}_{\{\tau>t\}}e^{\Gamma_t}Y|\mathscr{F}_t]. \quad (1.2)$$

Dimostrazione. Osservando che $\mathscr{H}_t \subseteq \mathscr{G}_t$ e che H_t è $\mathscr{H}_t$-misurabile e utilizzando le proprietà dell'attesa condizionata, abbiamo:

$$\mathbb{P}(\{\tau > t\}|\mathscr{G}_t) = E[\mathbb{1}_{\{\tau>t\}}|\mathscr{G}_t] = \mathbb{1}_{\{\tau>t\}}. \quad (1.3)$$

La tesi segue dall'uguaglianza dimostrata nel lemma precedente.
La seconda uguaglianza segue banalmente dalla definizione di $\Gamma_t = -\ln(G_t)$.
□

Lemma 1.1.3. *Sia Y una variabile aleatoria su uno spazio di probabilità $(\Omega, \mathscr{G}, \mathbb{P})$ e sia $t < T$. Allora:*

$$E[\mathbb{1}_{\{\tau>T\}}Y|\mathscr{G}_t] = \mathbb{1}_{\{\tau>t\}}E[\mathbb{1}_{\{\tau>T\}}e^{\Gamma_t}Y|\mathscr{F}_t]. \quad (1.4)$$

Se Y è $\mathscr{F}$-misurabile, allora

$$E[\mathbb{1}_{\{\tau>T\}}Y|\mathscr{G}_t] = \mathbb{1}_{\{\tau>t\}}E[e^{\Gamma_t-\Gamma_T}Y|\mathscr{F}_t]. \quad (1.5)$$

Dimostrazione. Osserviamo che $\mathbb{1}_{\{\tau>T\}}\mathbb{1}_{\{\tau>t\}} = \mathbb{1}_{\{\tau>T\}}$. Quindi la (1.4) segue banalmente dalla (1.2). Per la (1.5) abbiamo che, dalla (1.4):

$$\begin{aligned}
E\left[\mathbb{1}_{\{\tau>T\}}Y|\mathscr{G}_t\right] =& \mathbb{1}_{\{\tau>t\}}E\left[\mathbb{1}_{\{\tau>T\}}e^{\Gamma_t}Y|\mathscr{F}_t\right] = \\
=& \mathbb{1}_{\{\tau>t\}}E\left[E\left[\mathbb{1}_{\{\tau>T\}}|\mathscr{F}_T\right]e^{\Gamma_t}Y|\mathscr{F}_t\right] = \\
=& \mathbb{1}_{\{\tau>t\}}E\left[\mathbb{P}(\tau > T|\mathscr{F}_T)e^{\Gamma_t}Y|\mathscr{F}_t\right] = \\
=& \mathbb{1}_{\{\tau>t\}}E\left[G_T e^{\Gamma_t}Y|\mathscr{F}_t\right] = \\
=& \mathbb{1}_{\{\tau>t\}}E\left[e^{\Gamma_t-\Gamma_T}Y|\mathscr{F}_t\right].
\end{aligned}$$

□

Lemma 1.1.4. *Siano X e Y due variabili aleatorie su uno spazio di probabilità $(\Omega, \mathscr{G}, \mathbb{P})$. Sia $\mathscr{F} \subseteq \mathscr{G}$ una sotto σ-algebra tale che:*

- *X è indipendente da $\mathscr{F}$;*

- Y *è* $\mathscr{F}-$*misurabile.*

Sia $\mathscr{B}$ *la* $\sigma-$ *algebra dei Boreliani. Allora per ogni funzione* h $\mathscr{B}-$*misurabile e limitata si ha che:*

$$E[h(X,Y)|\mathscr{F}] = g(Y)$$

dove $g(h) = E[h(X,y)]$.

Dimostrazione. Bisogna provare che $g(Y)$ è una versione dell'attesa condizionata di $h(X,Y)$ rispetto alla $\sigma-$algebra $\mathscr{G}$ e cioè che

$$\int_G h(X,Y)d\mathbb{P} = \int_G g(Y)d\mathbb{P}$$

per ogni $G \in \mathscr{F}$. Indichiamo con $\mathbb{P}^W$ la distribuzione rispetto ad una variabile aleatoria W. Abbiamo che

$$g(y) = \int_{\mathbb{R}} h(x,y)\mathbb{P}^X(dx).$$

Essendo g misurabile rispetto alla $\sigma-$algebra dei Boreliani, $g(Y)$ è misurabile rispetto alla $\sigma-$algebra $\mathscr{G}$. Quindi, dato che $G \in \mathscr{F}$ e ponendo $\mathbb{Z} = \mathbb{1}_G$, si ha che

$$\begin{aligned}
\int_G h(X,Y)d\mathbb{P} &= \int_G h(X,Y)\mathbb{Z}d\mathbb{P} = \\
&= \int\int\int h(x,y)z\mathbb{P}^{(X,Y,Z)}(d(x,y,z)) = \\
&= \int\int\int h(x,y)z\mathbb{P}^{(X)}(d(x))\mathbb{P}^{(Y,Z)}(d(y,z)) = \\
&= \int\int g(y)z\mathbb{P}^{(Y,Z)}(d(y,z)) = \\
&= \int_G g(Y)d\mathbb{P}.
\end{aligned}$$

Nella terza disuguaglianza abbiamo usato l'indipendenza di X da $\mathscr{F}$ e nella quarta il teorema di Fubini. □

1.2 Tempo di default

Sia $(\Omega, \mathscr{G}, \mathbb{P}, (\mathscr{G}_t)_{t\geq 0})$ uno spazio di probabilità con filtrazione, e siano:

- $W = (W_t)_{t\geq 0}$ moto Browniano;
- ε una variabile casuale con distribuzione esponenziale di parametro $\lambda = 1$ indipendente da W;
- X processo definito dall'equazione differenziale stocastica

$$X_t = \mu(t, X_t)dt + \sigma(t, X_t)dW_t;$$

- $\gamma : [0, +\infty[\times\mathbb{R} \to \mathbb{R}$ una funzione non negativa e misurabile.

Consideriamo $\tau : \Omega \to \mathbb{R}$ una variabile aleatoria non negativa sullo spazio di probabilità $(\Omega, \mathscr{G}, \mathbb{P})$, così definita:

$$\tau = \inf\{t \geq 0 | \int_0^t \gamma(s, X_s)ds \geq \varepsilon\}. \tag{1.6}$$

Vogliamo dimostrare che per τ definito dalla equazione (5.1) vale:

- $\mathbb{P}(\tau = 0) = 0$;
- $\mathbb{P}(\tau > t) > 0$.

Prima di tutto abbiamo che $\mathbb{P}(\tau = 0) = \mathbb{P}(0 \geq \varepsilon) = 0$.

Lemma 1.2.1. *Sia τ il tempo di default appena definito. Allora:*

$$\mathbb{P}(\tau > t) > 0$$

Dimostrazione. Per dimostrare questa disuguaglianza, è sufficiente applicare il lemma (1.1.4). Essendo $\{\tau > t\} = \{\int_0^t \gamma(s, X_s)ds \leq \varepsilon\}$, risulta:

$$\begin{aligned}\mathbb{P}(\tau > t) =& \mathbb{P}(\int_0^t \gamma(s, X_s)ds < \varepsilon) = \\ =& E[\mathbb{1}_{\{\int_0^t \gamma(s,X_s)ds<\varepsilon\}}] = \\ =& E[E[\mathbb{1}_{\{\int_0^t \gamma(s,X_s)ds<\varepsilon\}}|\mathscr{F}_t]] = \\ =& E[\mathbb{P}(\int_0^t \gamma(s, X_s)ds < \varepsilon|\mathscr{F}_t)].\end{aligned}$$

Sia $h(x) := \mathbb{P}(\varepsilon > x)$, dove ε è indipendente da $\mathbf{F}$ e ha distribuzione esponenziale. Essendo $\{t \geq 0|\int_0^t \gamma(s, X_s)ds < \varepsilon\}$ $\mathscr{F}_t-$misurabile, si può applicare il lemma (1.1.4):

$$\mathbb{P}(\tau > t) = E[h(\int_0^t \gamma(s, X_s)ds)] > 0$$

. □

Inoltre abbiamo che:

$$\mathbb{P}(\tau > t|\mathscr{F}_t) = \mathbb{P}(\int_0^t \gamma(s, X_s))ds < \varepsilon|\mathscr{F}_t) = e^{-\int_0^t \gamma(s,X_s)ds}. \tag{1.7}$$

Quindi il processo $G_t = (\tau > t|\mathscr{F}_t)$ è positivo per ogni $t \in \mathbb{R}^+$, e possiamo pertanto definire il processo

$$\Gamma_t = -\ln(G_t) \quad \forall t \in \mathbb{R}^+.$$

Notiamo che, essendo $G_T = e^{-\Gamma_t}$, dalla (1.7) otteniamo:

$$\Gamma_t = \int_0^t \gamma(s, X_s)ds, \tag{1.8}$$

Definizione 1.9. Dalla (1.5), segue che:

$$\begin{aligned}\mathbb{P}(\tau > T|\mathscr{G}_t) = E[\mathbb{1}_{\{\tau>T\}}|\mathscr{G}_t] = \mathbb{1}_{\tau>t}E[e^{\Gamma_t-\Gamma_T}|\mathscr{F}_t] = \\ = \mathbb{1}_{\{\tau>t\}}E[e^{-\int_t^T \gamma(s,X_s)ds}|\mathscr{F}_t].\end{aligned}$$

definiamo la survival probability al tempo T, $Q(t, \tau > T)$, come:

$$Q(t, \tau > T) := E[e^{-\int_t^T \gamma(s,X_s)ds}|\mathscr{F}_t].$$

Essa, intuitivamente, rappresenta la probabilità che il default avviene dopo il tempo T, con le informazioni note al tempo t.

Abbiamo quindi definito un tempo di default τ che ci permette di introdurre una filtrazione $\mathbf{G} = \mathbf{H} \vee \mathbf{F}$ sullo spazio di probabilità $(\Omega, \mathscr{G}, \mathbb{P})$. Sullo spazio $(\Omega, \mathscr{G}, \mathbb{P}, (\mathscr{G}_t)_{t\geq 0})$ costruiremo il nostro modello, grazie a cui saremo in grado di valutare il prezzo di un'opzione, tenendo conto del rischio di credito.

Capitolo 2

Modelli a volatilità stocastica con default

Prima di descrivere il modello che prenderemo in considerazione per i nostri studi, elenchiamo alcune ipotesi di mercato su cui questo verrà costruito. Alcune di queste ipotesi non esistono in mercati reali, ma sono necessarie per semplificare il modello matematico.

- Prima di tutto è valido il principio di non arbitraggio, ovvero l'impossibilità di realizzare un guadagno certo senza alcun impegno di fondi. Espresso in maniera più rigorosa, il principio di assenza di arbitraggio dice: *Se due strumenti forniscono all'epoca finale T lo stesso risultato, allora, all'epoca iniziale t, devono avere lo stesso prezzo.*

- La misura di probabilità utilizzata è quella neutrale al rischio, cioè una misura di probabilità sotto la quale il prezzo di un'attività finanziaria è pari al suo valore atteso futuro scontato al tasso privo di rischio. Essa è anche nota come misura a martingala equivalente (EMM). Una misura a martingala equivalente Q è una misura di probabilità su uno spazio $(\Omega, \mathscr{F})$ tale che:

1. Q e P sono due misure equivalenti;

2. $\widetilde{S}_t = e^{-\int_0^t r_s ds} S_t$, $t \in [0,T]$ è una Q-martingala e in particolare si ha che:
$$S_t = E^Q[e^{-\int_t^T r_s ds} S_T | \mathscr{F}_t^W]$$
.

- Si suppone di essere in un tipo di mercato senza frizioni, ovvero in cui non ci siano costi di transazioni.

2.1 Modello di mercato con tasso d'interesse non costante

Consideriamo lo spazio di probabilità con filtrazione, definito nel primo capitolo, $(\Omega, \mathscr{G}, \mathbb{P}, (\mathscr{G}_t)_{\{t\geq 0\}})$, dove $\mathbf{G} = \mathbf{F} \vee \mathbf{D}$ ed $\mathbf{F}$ è la filtrazione generata dal processo X_t definito di seguito. Sia S un bene finanziario (asset) le cui dinamiche sono definite dalle seguenti equazioni:

$$\begin{aligned} &S_t = \mathbb{1}_{\{\tau > t\}} e^{X_t}, \\ &dX_t = \mu(t, X_t, r_t)dt + \sigma(t, X_t, r_t)dW_t^1, \\ &dr_t = \alpha(t, X_t, r_t)dt + \beta(t, X_t, r_t)dW_t^2, \\ &\rho(t, X_t, r_t)dt = d < W^1, W^2 >_t, \\ &\tau = \inf\{t \geq 0 | \int_0^t \gamma(s, X_s)ds \geq \varepsilon\}, \end{aligned} \tag{2.1}$$

dove

- $W^1 = (W_t^1)_{t\geq 0}$ e $W^2 = (W_t^2)_{t\geq 0}$ sono due moti Browniani reali correlati;

- ε è una variabile casuale con distribuzione esponenziale di parametro $\lambda = 1$, indipendente da $W = (W^1, W^2)$;

- $\tau : \Omega \to \mathbb{R}$ è il tempo di default di S_t;

- $|\rho| < 1$.

Le equazioni definite dalla (2.1) includono genericamente tutti i modelli di voaltilità locale, a seconda della scelta delle funzioni α, β, γ, σ e μ. In questa tesi tratteremo un particolare modello per prezzare opzioni di tipo Europeo. Enunciamo dei risultati preliminari che hanno una valenza generale e che in seguito verranno applicati al particolare modello oggetto di studio.

Proposizione 2.1.1. *Se vale il principio di non arbitraggio, si ha che*

$$\mu(t, X_t, r_t) = r_t(t, X_t, r_t) + \gamma(t, X_t) - \frac{1}{2}\sigma^2(t, X_t, R_t).$$

Dimostrazione. Poiché è vera l'ipotesi EMM e vale il principio di non arbitraggio, si ha che $\widetilde{S}_t = e^{-\int_0^t r_s ds} S_t$ è una $\mathbf{G}-$martingala e quindi:

$$e^{-\int_0^t r_s ds} S_t = E[e^{-\int_0^T r_s ds} S_T | \mathscr{G}_t]$$

$$\Updownarrow$$

$$S_t = E[e^{-\int_t^T r_s ds} S_T | \mathscr{G}_t]$$

$$\Updownarrow$$

$$\mathbb{1}_{\{\tau>t\}} e^{X_t} = E[e^{-\int_t^T r_s ds} \mathbb{1}_{\{\tau>t\}} e^{X_T} | \mathscr{G}_t]$$

$$\Updownarrow$$

$$\mathbb{1}_{\{\tau>t\}} e^{X_t} = E[e^{-\int_t^T r_s ds + X_T} \mathbb{1}_{\{\tau>t\}} | \mathscr{G}_t]$$

$$\Updownarrow$$

$$\mathbb{1}_{\{\tau>t\}} e^{X_t} = \mathbb{1}_{\{\tau>t\}} E[e^{-\int_t^T r_s ds + X_T} e^{\Gamma_t - \Gamma_T} | \mathscr{F}_t]$$

$$\Updownarrow$$

$$e^{X_t - \int_0^t r_s ds - \int_0^t \gamma(s, X_s) ds} = E[e^{X_t - \int_0^T r_s ds - \int_0^T \gamma(s, X_s) ds} | \mathscr{F}_t].$$

Ne segue che $\widetilde{S}_t = e^{-\int_0^t r_s ds} S_t$ è una $\mathbf{G}-$martingala se e solo se $Y_t := e^{X_t - \int_0^t r_s ds - \int_0^t \gamma(s,X_s) ds}$ è una $\mathbf{F}-$martingala.
Applicando la formula di Ito a Y_t risulta:

$$\begin{aligned} dY_t =& Y_t(-r_t(t,X_t,r_t) - \gamma(t,X_t,r_t))dt + Y_t dX_t + \frac{1}{2} Y_t d < X >_t = \\ =& Y_t(-r_t(t,X_t,r_t) - \gamma(t,X_t,r_t))dt + Y_t \mu(t,X_t,r_t)dt + \\ &+ Y_t \sigma(t,X_t,r_t)dW_t + \frac{1}{2} Y_t \sigma^2(t,X_t,r_t)dt = \\ =& Y_t(-r_t(t,X_t,r_t) - \gamma(t,X_t,r_t) + \mu(t,X_t,r_t) + \\ &+ \frac{1}{2} Y_t \sigma^2(t,X_t,r_t))dt + Y_t \sigma(t,X_t,r_t)dW_t. \end{aligned}$$

Quindi ponendo $-r_t(t,X_t,r_t) - \gamma(t,X_t,r_t) + \mu(t,X_t,r_t) + \frac{1}{2} Y_t \sigma^2(t,X_t,r_t) = 0$ si ha

$$\mu(t,X_t,r_t) = r_t(t,X_t,r_t) + \gamma(t,X_t) - \frac{1}{2}\sigma^2(t,X_t,r_t)$$

. □

Per garantire che $\widetilde{S}_t$ sia una martingala dobbiamo imporre delle restrizioni sulla funzione μ, μ è determinata da $\sigma(t,X_t,r_t)$, $\gamma(t,X_t)$, $r(t,X_t,r_t)$, dove

- r_t è il tasso d'interesse privo di rischio;
- $\sigma^2(t,X_t,r_t)$ è la voaltilità locale;
- $\gamma(t,X_t)$ è la funzione di default.

Possiamo quindi riscrivere le equazioni che determinano il modello con default nel seguente modo:

$$\begin{aligned} &S_t = 1_{\{\tau > t\}} e^{X_t}, \\ &dX_t = (r_t(t,X_t,r_t) + \gamma(t,X_t) - \frac{1}{2}\sigma^2(t,X_t,r_t))dt + \sigma(t,X_t,r_t)dW_t^1, \\ &dr_t = \alpha(t,X_t,r_t)dt + \beta(t,X_t,r_t)dW_t^2, \\ &\rho(t,X_t,r_t)dt = d < W^1, W^2 >_t, \\ &\tau = \inf\{t \geq 0 | \int_0^t \gamma(s,X_s)ds \geq \varepsilon\}. \end{aligned} \tag{2.2}$$

2.2 Valutazione delle opzioni

Consideriamo un'opzione di tipo europeo con data di scadenza T. Tenendo conto della possibilità di default e delle ipotesi di mercato, il suo prezzo è dato dalla seguente equazione:

$$E[e^{-\int_0^T r_s ds} H(S_T)|\mathscr{G}_t] \tag{2.3}$$

dove $H(S_T)$ è il payoff di S al tempo T. Il nostro obiettivo è quindi quello di calcolare (2.3). Con i risultati ottenuti nel primo capitolo, sappiamo come esprimere (2.3) in termini di $\mathscr{F}_t$ invece che in termini di $\mathscr{G}_t$.

Teorema 2.2.1. *Sia V_t il prezzo di S al tempo t, H il payoff di S in caso in cui non avvenga default prima di T. Allora*

$$V_t = K + \mathbb{1}_{\{\tau>t\}} E[e^{-\int_t^T r_s+\gamma(s,X_s)ds}(h(X_T) - K)|\mathscr{F}_t] \tag{2.4}$$

dove

$$h(x) := H(e^x), H(0) = K.$$

Dimostrazione. Abbiamo due possibilità:

1. se $\omega \in \{\tau > T\}$, allora $S_t = e^{X_t} \Rightarrow H(S_t) = H(e^{X_t}) = h(X_t)$;
2. se $\omega \in \{\tau \leq T\}$, allora $S_t = 0 \Rightarrow H(S_t) = H(0) = h(0) = K$.

$$\begin{aligned}
V_t =& E[e^{-\int_t^T r_s ds} H(S_T)|\mathscr{G}_t] = \\
=& E[e^{-\int_t^T r_s ds} \mathbb{1}_{\{\tau>T\}} h(X_T)|\mathscr{G}_t] + KE[e^{-\int_t^T r_s ds} \mathbb{1}_{\{\tau\leq T\}}|\mathscr{G}_t] = \\
=& E[e^{-\int_t^T r_s ds} \mathbb{1}_{\{\tau>T\}} h(X_T)|\mathscr{G}_t] + K - KE[e^{-\int_t^T r_s ds} \mathbb{1}_{\{\tau>T\}}|\mathscr{G}_t] = \\
=& E[e^{-\int_t^T r_s ds} \mathbb{1}_{\{\tau>T\}}(h(X_T) - K)|\mathscr{G}_t] + K = \\
=& K + \mathbb{1}_{\{\tau>t\}} E[e^{-\int_t^T r_s ds} e^{-\int_t^T \gamma(s,X_s)ds}(h(X_T) - K)|\mathscr{F}_t] = \\
=& K + \mathbb{1}_{\{\tau>t\}} E[e^{-\int_t^T (r_s+\gamma(s,X_s))ds}(h(X_T) - K)|\mathscr{F}_t].
\end{aligned}$$

□

Dalla formula (2.4) segue quindi che per valutare il prezzo di un'opzione, sarà necessario calcolare formule del tipo

$$u(t,x,r) = E[e^{-\int_t^T r_s+\gamma(s,X_s)ds} h(X_T)|X_t = x, r_t = r]. \tag{2.5}$$

Se applichiamo il teorema di Feynman-Kač, la funzione $u(t,x,r)$ definita dalla (2.4), può essere vista come soluzione del seguente problema di Cauchy

$$\begin{cases} (\mathscr{A} + \partial_t)u(t,x,r) = 0 \\ u(T,x,r) = \phi(x,r) \end{cases} \tag{2.6}$$

Dove $\mathscr{A}$ è l'operatore definito di seguito

$$\mathscr{A}(t,x,r) = \frac{1}{2}\sigma^2(t,x,r)\partial_{xx} + \frac{1}{2}\beta^2(t,x,r)\partial_{rr} +$$

$$+\mu(t,x,r)\partial_x + \alpha(t,x,r)\partial_r + \rho(t,x,r)\sigma(t,x,r)\beta(t,x,r)\partial_{x,r} - \gamma(t,x,r) - r.$$

2.3 Modelli JDCEV e di Vasiceck

Adesso consideriamo due modelli specifici per descrivere X_t e r_t, rispettivamente JDCEV (Jump to Default Constant Elasticity of Variance) e il modello di Vasicek. Il primo tiene conto, da un lato della volatilità e dell'effetto leva, e dall'altro della possibilità di default. É un'estensione del noto modello CEV (constant elasticity of variance), inizialmente introdotto come generalizzazione del modello Black-Scholes. Esso determinato dalla scelta della funzioni σ e γ dell'equazione differenziale stocastica di X_t:

$$\sigma(t,x,r) = \sigma e^{(\beta-1)x};$$
$$\gamma(t,x,r) = b + c\sigma^2(t,x,r) = b + ce^{2(\beta-1)x},$$

con

- $b \geq 0$,
- $c \geq 0$,

- $\sigma > 0$,
- $0 \leq \beta \leq 1$.

Notiamo che, affinché l'esponente della funzione σ sia negativo, è necessario richiedere che $\beta < 1$. Questo vincolo è importante dal punto di vista finanziario perché fa sì che σ cresca quando $S \to 0^+$, che è un risultato coerente con l'ipotesi che dell'effetto leva.

Il modello di Vasicek descrive il tasso d'interesse privo di rischio r_t ponendo:

$$\alpha(t, x, r) = k(\theta - r),$$
$$\beta(t, x, r) = \delta,$$

con

- $k > 0$,
- $\theta > 0$,
- $\delta > 0$.

Cioè r_t è descritto dalla seguente equazione differenziale stocastica:

$$dr_t = k(\theta - r_t)dt + \delta dW_t^2$$

Ricordando che $\mu(t, x, r) = r_t(t, X_t, r_t) + \gamma(t, X_t) - \frac{1}{2}\sigma^2(t, X_t, r_t)$, e combinando il JDCEV con il Vasiceck, otteniamo:

$$\begin{aligned}
&S_t = \mathbb{1}_{\{\tau > t\}} e^{X_t} \\
&dX_t = \left(r_t + b + (c - \frac{1}{2})\sigma^2 e^{2(\beta-1)X_t} \right) dt + \sigma e^{(\beta-1)X_t} dW_t^1 \\
&dr_t = k(\theta - r_t)dt + \delta dW_t^2 \\
&\rho dt = d < W^1, W^2 >_t \\
&\tau = \inf\{t \geq 0 | \int_0^t b + ce^{2(\beta-1)X_s} ds \geq \varepsilon\}
\end{aligned} \tag{2.7}$$

Se scegliamo $|\rho| < 1$ costante, i moti Browniani correlati sono descritti dal seguente sistema di equazioni:

$$\begin{cases} W_t^1 = \hat{W}_t^1 \\ W_t^2 = \rho W_t^1 + \sqrt{1-\rho^2}\hat{W}_t^2 \end{cases}$$

dove $\hat{W}_t = (\hat{W}_t^1, \hat{W}_t^2)$ è un moto browniano bidimensionale.

2.4 Zero Coupon Bond

Consideriamo un'obbligazione che abbia payoff 1 al tempo T se non avviene nessun default prima della data di scadenza T, 0 altrimenti. Allora applicando il teorema (2.2.1), ponendo $K = 0$ e $h(X_T) = 1$, il prezzo di tale obbligazione è

$$V_t = \mathbb{1}_{\{\tau > t\}} E[e^{-\int_t^T r_s + \gamma(s, X_s) ds} | \mathscr{F}_t].$$

Poniamo

$$u(t, X_t, r_t, T) := E[e^{-\int_t^T r_s + \gamma(s, X_s) ds} | \mathscr{F}_t],$$

allora si ha che

$$V_t = \mathbb{1}_{\{\tau > t\}} u(t, X_t, r_t, T).$$

Calcolare il valore di $u(t, X_t, r_t, T)$ è il nostro obiettivo, ma per poterlo fare è necessario ricorrere a dei metodi numerici. Ci concentreremo in particolar modo sul metodo Monte Carlo, descritto nel seguente capitolo.

Capitolo 3

Metodi Monte Carlo

Come abbiamo visto nel secondo capitolo, calcolare il prezzo di un'opzione potrebbe ridursi a calcolare il valore atteso di una variabile aleatoria. Nella maggior parte dei casi, non si è in grado di trovare una formula esplicita che ci permetta di calcolare analiticamente questi valori attesi. Per questo motivo introduciamo il metodo Monte Carlo, uno dei metodi numerici più importanti per il calcolo del valore atteso di una variabile aleatoria. Esso è basato sulla legge dei grandi numeri che enunciamo di seguito.

Teorema 3.0.1. *Sia (X_n) una sequenza di variabili aleatorie indipendenti e identicamente distribuite su uno spazio di probabilità $(\Omega, \mathscr{F}, \mathbb{P})$, tali che $E[X_1] < \infty$. Allora:*

$$\lim_{n\to+\infty} \frac{1}{n} \sum_{i=1}^{n} X_i = E[X_1] \quad q.s.$$

Questo significa che, se vogliamo calcolare il valore atteso di una variabile aleatoria X e siamo in grado di generare $\widetilde{X_1}, \widetilde{X_2}, \ldots, \widetilde{X_n}$ realizzazioni indipendenti di X, possiamo approssimare quasi sicuramente $E[X]$ con

$$\frac{1}{n} \sum_{i=1}^{n} \widetilde{X_i}.$$

Per capire che errore introduciamo con questa approssimazione, ricordiamo la disuguaglianza di Markov.

Teorema 3.0.2. *Sia X una variabile aleatoria su uno spazio di probabilità $(\Omega, \mathscr{F}, \mathbb{P})$. Siano $\lambda > 0 \in \mathbb{R}$, $1 \leq p < \infty$. Allora:*

$$\mathbb{P}(|X| \geq \lambda) \leq \frac{E[|X|^p]}{\lambda^p}.$$

In particolare, se X è integrabile, vale:

$$\mathbb{P}(|X - E[X]| \geq \lambda) \leq \frac{var(X)}{\lambda^2}.$$

Applicando la disuguaglianza di Markov a $M_n = \frac{1}{n}\sum_{i=1}^{n} X_i$ e ponendo $\mu = E[X_1]$, abbiamo che:

$$\begin{gathered}\mathbb{P}(|M_n - \mu| \geq \varepsilon) \leq \frac{var(M_n)}{\varepsilon^2} = \\ = \frac{var(\frac{1}{n}\sum_{i=1}^{n} X_i)}{\varepsilon^2} = \\ = \frac{\frac{1}{n^2} var(\sum_{i=1}^{n} X_i)}{\varepsilon^2} = \\ = \frac{\frac{1}{n} var(X_1)}{\varepsilon^2} = \\ = \frac{var(X_1)}{n\varepsilon^2}.\end{gathered}$$

Quindi, ponendo $\sigma^2 := var(X)$:

$$\mathbb{P}(|M_n - \mu| \geq \varepsilon) \leq \frac{\sigma^2}{n\varepsilon^2},$$

$$\Downarrow$$

$$\mathbb{P}(|M_n - \mu| < \varepsilon) \geq 1 - \frac{\sigma^2}{n\varepsilon^2} =: p$$

Abbiamo quindi trovato una stima dell'errore in termini di:

- n, il numero di simulazioni di X;
- ε, errore di approssimazione massimo;

- p, la probabilità massima che il valore approssimato M_n appartenga all'intervallo $[\mu - \varepsilon, \mu + \varepsilon]$.

Ne segue che, fissato $n \in \mathbb{N}$ e $p \in]0, 1[$, l'errore di approssimazione massimo del metodo Monte Carlo è

$$\varepsilon = \frac{\sigma}{\sqrt{n(1-p)}}.$$

Quindi, indipendentemente dalla dimensione del problema, l'errore tende a zero come $\frac{1}{\sqrt{n}}$.
Tipicamente il valore di σ non è noto, ma è possibile utilizzare una sua approssimazione:

$$\sigma_n^2 = \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \mu_n)^2, \qquad \mu_n := \frac{1}{n}\sum_{i=1}^{n} X_i.$$

In altre parole, con il metodo Monte Carlo, possiamo generare delle realizzazioni di X sia per ottenere un'approssimazione del valore atteso $E[X]$, sia per avere una stima dell'errore che commettiamo. In particolare indichiamo con δ il livello di confidenza, e cioè $1 - \delta$ è la probabilità che il valore stimato cada nell'intervallo

$$[\mu_n - \frac{z_\delta \sigma_n}{\sqrt{n}}, \mu_n + \frac{z_\delta \sigma_n}{\sqrt{n}}],$$

dove z_δ è tale che

$$\int_{-\infty}^{z_\delta} \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} = \frac{\delta}{2}.$$

Se fissiamo $1 - \delta = 99\%$, allora $z_\delta = 2.58$.

3.1 Generazione di numeri casuali

Come abbiamo visto nel primo paragrafo, alla base del metodo Monte Carlo, c'è la generazione di numeri casuali. Il solo fatto che dei numeri casuali vengano generati tramite un algoritmo però, fa sì che questi non siano realmente casuali. Si parla in realtà di numeri *pseudo-casuali*. E' quindi importante trovare dei *buoni* generatori di numeri pseudo-casuali, ovvero dei

generatori in grado di costruire una sequenza di numeri, difficilmente distinguibile da una reale sequenza di numeri casuali.
Di seguito descriveremo due algoritmi fondamentali: il primo che permette di generare numeri pseudo-casuali uniformemente distribuiti nell'intervallo $[0, 1]$; il secondo che permette, a partire da un insieme di numeri pseudo-casuali uniformemente distribuiti nell'intervallo $[0, 1]$, di generare una sequenza di numeri con distribuzione normale, di media 0 e varianza 1.
Questi due algoritmi verranno poi utilizzati per poter applicare il metodo Monte Carlo al modello presentato nei capitolo precedenti.

3.1.1 Numeri pseudo-casuali con distribuzione uniforme

Prima di tutto è necessario specificare cosa intendiamo per generatore *genuino* di numeri casuali. Esso è un algoritmo in grado di produrre una sequenza di numeri $U_1, U_2, \ldots, U_k$ che verifichino le due seguenti proprietà:

(i) ogni U_i è uniformemente distribuita nell'intervallo $]0, 1[$;

(ii) i valori U_i sono reciprocamente indipendenti.

La proprietà (i) è una normalizzazione conveniente ma allo stesso arbitraria in quanto variabili casuali uniformemente distribuite possono essere facilmente trasformate in variabili con una differente distribuzione, come vedremo nel prossimo paragrafo; la (ii) invece è più importante: implica che i valori U_i siano a due a due indipendenti e, in particolare, che U_i non è predicibile da $U_1, U_2, \ldots, U_{i-1}$.
Naturalmente la seconda proprietà non potrà mai essere verificata, ma sarà sufficiente generare una sequenza di numeri casuali $u_1, \ldots, u_k$ tali che, per piccoli valori di k, tale sequenza sarà indistinguibile da una vera sequenza di variabili uniformemente distribuite. Introduciamo a tal fine, l'algoritmo chiamato *generatore lineare congruenziale*, definito ricorsivamente, a partire

da un fissato x_0 in questo modo:

$$\begin{cases} x_{i+1} = (ax_i + c) \, mod \, m \\ u_{i+1} = \frac{x_{i+1}}{m}, \end{cases} \tag{3.1}$$

dove i parametri a, b, c sono interi.
Una definizione importante da introdurre è quella di periodo del generatore, ovvero il numero di valori distinti che esso produce, prima che questi inizino a ripetersi. Per come è definito il generatore lineare congruenziale, i numeri generati e, conseguentemente, il periodo, dipendono dal valore iniziale x_0. Diremo che il generatore ha periodo *pieno*, se, a partire da un qualunque valore di x_0, esso ha periodo massimo (che si dimostra essere $m - 1$).
E' possibile trovare delle condizioni sotto cui, il generatore ha periodo pieno. Se $c \neq 0$, le condizioni solo le seguenti:

- a ed m sono numeri relativamente primi;

- ogni primo che divide m, divide anche $a - 1$;

- se m è divisibile per 4, lo è anche $a - 1$.

Se invece $c = 0$ e m è un numero primo, il periodo è massimo per qualunque valore di x_0 se:

- a^{m-1} è un multiplo di m;

- a^{j-1} non è un multiplo di m, per ogni $j = 1, \ldots, m - 2$.

La differenza sostanziale tra i due casi, è che , ponendo $c = 0$, l'algoritmo è più veloce.
Notiamo che, se la memoria di macchina massima per un intero è 32 bit, m può essere al più $2^{31} - 1$ (considerando che un bit è occupato dal segno). E' possibile però trovare degli algoritmi efficienti che abbiano periodo più

grande. Uno di questi è il generatore lineare congruenziale combinato. Come si può intendere già dal nome, si prendono J differenti generatori lineari congruenziali, dopo aver fissato $x_1, \ldots, x_J$ valori iniziali, ciascuno con parametri a_m, m_j e $c_j = 0$:

$$\begin{cases} x_{i+1}^j = (a_j x_i) \; mod \, m_j \\ u_{i+1}^j = \frac{x_{i+1}^j}{m_j}, \quad j = 1, \ldots, J, \end{cases} \tag{3.2}$$

e si pone:

$$\begin{cases} x_{i+1} = \sum_{j=1}^{J} (-1)^{j-1} u_{i+1}^j \; mod \, m_j \\ u_{i+1} = \frac{x_{i+1}}{m}, \quad se \; x_{i+1} > 0 \\ u_{i+1} = \frac{m-1}{m}, \quad se \; x_{i+1} = 0, \end{cases} \tag{3.3}$$

dove $m := \max_j m_j$.

Un altro algoritmo importante è il generatore ricorsivo multiplo (MRG) così definito:

$$\begin{cases} x_i = (a_1 x_{i-1} + a_2 x_{i-2} + \ldots + a_k x_{i-k}) \; mod \, m \\ u_i = \frac{x_i}{m}, \end{cases} \tag{3.4}$$

dove $i > k$, e a_i e m sono dei parametri interi. In questo caso, è necessario fissare k valori iniziali, $x_1, \ldots, x_k$.

E' possibile combinare degli algoritmi MRG proprio come descritto dalle equazioni (3.3).

Riportiamo di seguito il codice scritto in $C++$ che implementa un generatore ricorsivo multiplo combinato, con

- $k = 3$,
- $J = 2$,
- $m_1 = 2^{31} - 1 = 2147483647$,
- $a_1^1 = 0$,
- $a_2^1 = 63308$,

- $a_3^1 = -183326$,
- $m_2 = 2145483479$,
- $a_1^2 = 86098$,
- $a_2^2 = 0$,
- $a_3^2 = -539608$.

Il periodo di questo generatore è circa 2^{185}, che è un numero sorprendente: questo algoritmo genera una sequenza di circa $4.9040 \cdot 10^{55}$ numeri distinti, prima che la sequenza si ripeta.

MRG in $C++$

```
#include <iostream> #include <string.h>
#include <math.h>
#include <ctime>
#include <cstdlib>
using namespace std;
#define m1 2147483647
#define m2 2145483479
#define a12 63308
#define a13 .183326
#define a21 86098
#define a23 .539608
#define q12 33921
#define q13 11714
#define q21 24919
#define q23 3976
#define r12 12979
#define r13 2883
#define r21 7417
#define r23 2071
#define Invmp1 0.00000000046566612873077393
int x10=7, x11=8, x12=9, x20=10, x21=11, x22=12;
int Random(){
int h, p12, p13, p21, p23,i;
h=x10/q13; p13=-a13*(x10-h*q13)-h*r13;
h=x11/q12; p12=a12*(x11-h*q12)-h*r12;
if(p13<0) p13=p13+m1; if(p12<0) p12=p12+m1;
x10=x11;x11=x12;x12=p12-p13; if(x12<0) x12=x12+m1;
h=x20/q23; p23=-a23*(x20-h*q23)-h*r23;
h=x22/q21; p21=a21*(x22-h*q21)-h*r21;
if(p23<0) p23=p23+m2; if(p21<0) p21=p21+m2;
if (x12<x22) return (x12-x22+m1);
else return (x12-x22);}
double Uniform01(){
int Z;
Z=Random(); if(Z==0) Z=m1; return (Z*Invmp1);}
```

L'algoritmo che è appena stato riportato, verrà utilizzato nel resto della tesi, per la generazione di numeri pseudo-casuali uniformemente distribuiti nell'intervallo $]0, 1[$.

3.1.2 Numeri pseudo-casuali con distribuzione normale

Supponiamo di voler adesso generare una sequenza di numeri pseudo-casuali con funzione di ripartizione nota, $F(x)$. Cioè vogliamo generare una variabile aleatoria X su uno spazio di probabilità $(\Omega, \mathscr{F}, \mathbb{P})$ tale che $\mathbb{P}(X \leq x) = F(x)$. Allora vale che:

$$X = F^{-1}(U), \quad U \sim Unif[0, 1]$$

dove F^{-1} è l'inversa della funzione di ripartizione F, e U è una variabile con distribuzione uniforme nell'intervallo $[0, 1]$. Questo implica che, una volta generata una sequenza di numeri uniformemente distribuiti in $[0, 1]$, per trasformarli in una sequenza di numeri con funzione di ripartizione F, è sufficiente conoscere F^{-1}.
Nella nostra tesi, la distribuzione oggetto di interesse è quella normale, con media 0 e varianza 1, vale a dire:

$$F(x) := \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{u^2}{2}} du.$$

Notiamo che per la simmetria della distribuzione gaussiana, si ha che

$$F^{-1}(1 - u) = -F^{-1}(u), \tag{3.5}$$

quindi sarà sufficiente calcolarla nell'intervallo [0.5, 1) e poi estenderla a tutto l'intervallo $(0, 1)$ con la (3.5). Calcolare F^{-1} non è semplicissimo, tuttavia Beasley e Springer ne hanno fornito un'approssimazione:

$$F(u) \approx \frac{\sum_{n=0}^{3} a_n (u - \frac{1}{2})^{2n+1}}{1 + \sum_{n=0}^{3} b_n (u - \frac{1}{2})^{2n}}, \quad 0.5 \leq u \leq 0.92$$

,

$$F(u) \approx \sum_{n=0}^{8} c_n[log(-log(1-u))]^n, \quad 0.92 \leq u < 1$$

con i parametri a_n, b_n, c_n così definiti:

$a_0 = 2.50662823884$ $b_1 = 23.08336743743$
$a1 = -18.61500062529$ $b1 = 23.08336743743$
$a2 = 41.39119773534$ $b2 = -21.06224101826$
$a3 = -25.44106049637$ $b3 = 3.13082909833$
$c0 = 0.3374754822726147$ $c5 = 0.0003951896511919$
$c1 = 0.9761690190917186$ c6= 0.0000321767881768
$c2 = 0.1607979714918209$ c7 = 0.0000002888167364
$c3 = 0.0276438810333863$ c8 = 0.0000003960315187
$c4 = 0.0038405729373609$

Riportiamo di seguito il codice scritto in $C++$ che implementa l'algoritmo di Beasley e Springer per calcolare l'inversa di F.

Algoritmo Beasley e Springer in $C++$

```
#include <iostream>
#include <string.h>
#include <math.h>
#include <ctime>
#include <cstdlib>
#include <fstream>
using namespace std;
#define a0  2.50662823884
#define b0  -8.47351093090
#define a1  -18.61500062529
#define b1  23.08336743743
#define a2  41.39119773534
#define b2 -21.06224101826
#define a3  -25.44106049637
#define b3  3.13082909833
#define c0  0.3374754822726147
#define c5  0.0003951896511919
#define c1  0.9761690190917186
#define c6  0.0000321767881768
#define c2  0.1607979714918209
#define c7  0.0000002888167364
#define c3  0.0276438810333863
#define c8  0.0000003960315187
#define c4  0.0038405729373609
double Inversa(double u){
double y,x,r;bool a=0;
if (u<0.5){
        u=1-u;
        a=1;}
y=u-0.5;
if((u)<=0.92&&u>=0.5){
        r=y*y;
        x=y*(((a3*r+a2)*r+a1)*r+a0)/((((b3*r+b2)*r+b1)*r+b0)*r+1);}
else{
        r=log(-log(1-u));
        x=c0+r*(c1+r*(c2+r*(c3+r*(c4+r*(c5+r*(c6+r*(c7+r*c8)))))));  }
if (a==1) x=-x;
return x;}
```

Gli algoritmi descritti fino ad ora sono necessari per l'implementazione del metodo Monte Carlo. Nel prossimo paragrafo, descriviamo in particolar modo l'algoritmo Monte Carlo per calcolare il valore atteso e spieghiamo quindi il motivo per cui è stato necessario parlare anche di generatori di numeri casuali con distribuzione normale.

3.2 Metodo Monte Carlo per per il nostro modello

La quantità che vogliamo calcolare utilizzando il metodo Monte Carlo è:

$$\mathbb{E}[e^{-\int_0^T (r_s+\gamma_s(X_s))ds}].$$

Ricordiamo che:

$$\begin{aligned}
&\gamma_s(X_s) = b + c\sigma^2 e^{2(\beta-1)X_s},\\
&dX_t = \left(r_t + b + (c-\frac{1}{2})\sigma^2 e^{2(\beta-1)X_t}\right)dt + \sigma e^{(\beta-1)X_t} dW_t^1,\\
&dr_t = k(\theta - r_t)dt + \delta dW_t^2.
\end{aligned}$$

Remark 1. Un moto Browniano reale su uno spazio di probabilità con filtrazione $\mathbf{F}$, è un processo $(W_t)_{t\geq 0}$ tale che:

- $W_0 = 0$ q.s,
- W è $\mathbf{F}$-adattato ed è continuo in t,
- per ogni $0 \leq t < s$, $W_s - W_t$ ha distribuzione normale $N_{0,s-t}$ ed è indipendente da $\mathscr{F}_t$. Ne segue banalmente che $W_t \sim N_{0,t}$.

Primo step:
Prima di tutto dobbiamo simulare dei percorsi del processo stocastico X_t. A tal fine, è necessario discretizzare l'intervallo $[0,1]$ in $M+1$ punti t_i, $t_0 = 0 < \ldots < t_M = T$ ed utilizzare utilizzare una formula iterativa per calcolare X_{t_i} per ogni $i = 0, \ldots, M$.

Dobbiamo calcolare N simulazioni differenti di r_t e di X_t e lo facciamo con il metodo di Eulero:

$$r^{(j)}_{t_{i+1}} = r^{(j)}_{t_i} + k(\theta - r^{(j)}_{t_i})(t_{i+1} - t_i) + \delta(W^2_{t_{i+1}} - W^2_{t_i}),$$
$$X^{(j)}_{t_{i+1}} = X^{(j)}_{t_i} + (r^{(j)}_{t_i} + b + (c - \frac{1}{2})\sigma^2 e^{2(\beta-1)X^{(j)}_{t_i}})(t_{i+1} - t_i) + \sigma e^{(\beta-1)X^{(j)}_{t_i}}(W^1_{t_{i+1}} - W^1_{t_i}),$$

per ogni $j = 1 \dots N$.
W^1 e W^2 sono moti Browniani dipendenti tra loro, così calcolati:

$$\mathrm{W}^1_{t_{i+1}} - W^1_{t_i} = \sqrt{t_{i+1} - t_i} \cdot Z^1_{t_i}, \quad Z^1_{t_i} \sim N(0,1) \quad i = 0...M$$
$$W^2_{t_{i+1}} - W^2_{t_i} = \sqrt{t_{i+1} - t_i} \cdot (\rho Z^1_{t_i} + \sqrt{1-\rho^2} Z^2_{t_i}), \quad Z^2_{t_i} \sim N(0,1) \quad i = 0...M$$

Queste due ultime equazioni spiegano il motivo per cui abbiamo descritto alcuni generatori di numeri pseudo-casuali con distrbuzione normale di media 0 e varianza 1.

Secondo step:
Dobbiamo calcolare gli integrali $I(j) := \int_0^T (r^{(j)}_s + \gamma_s(X^{(j)}_s))ds$, per ogni $j = 1...N$. Approssimiamo gli integrali $I(j)$ mediante le formule di Newton-Cotes:

$$\int_0^T f(t)dt \approx \frac{h}{3}(f(t_0) + 4f(t_1) + 2f(t_2) + \dots 4f(t_{n-1} + 2f(t_n))).$$

Terzo step:
Facciamo la seguente approssimazione:

$$\mathbb{E}[e^{-\int_0^T r_s + \gamma_s ds}] \approx \frac{1}{N}\sum_{j=1}^{N} e^{-I(j)}.$$

Prima di mostrare i risultati ottenuti con questo algoritmo, introduciamo nel seguente capitolo il calcolo in parallelo.

Capitolo 4

CUDA - Compute Unified Device Architecture

Arrivati a questo punto, c'è un grosso limite da tener presente per i metodi Monte Carlo: i tempi di esecuzione. Immaginiamo di voler risolvere un problema che necessiti della generazione di milioni e milioni di numeri pseudo casuali. L'algoritmo potrebbe essere estremamente lento nel produrre il risultato richiesto, a tal punto da rendere il metodo Monte Carlo inutilizzabile. Per questo motivo, descriviamo in questo capitolo un tipo di architettura hardware per l'elaborazione in parallelo, CUDA, che ci permetterà di superare questo ostacolo.

4.1 Introduzione al calcolo in parallelo

L'obiettivo principale della programmazione in parallelo, è quello di migliorare la velocità di calcolo. Da un punto di vista puramente computazionale, essa può essere definita come un tipo di programmazione in cui i calcoli vengono effettuati simultaneamente, operando sul principio che grandi problemi, spesso, possono essere divisi in problemi più piccoli, i quali a loro volta possono essere risolti contemporaneamente.
Come è possibile fare ciò? Supponiamo di avere risorse di calcolo multiple.

Il calcolo in parallelo prevede l'uso simultaneo di queste risorse (che siano core o computer), ciascuna delle quali risolve uno dei *piccoli* problemi di cui è composto il nostro problema iniziale.
Gli aspetti software e hardware della programmazione in parallelo sono strettamente connessi tra di loro. Da un lato la parte hardware deve supportare il parallelismo al livello di architettura, dall'altro la programmazione deve focalizzarsi sul modo in cui un problema può essere risolto parallelamente.
La componente chiave per una macchina di calcolo altamente performante è la *central processing unit (CPU)*, comunemente chiamata anche *core*. Essa costituisce l'elemento che caratterizza le prestazioni dell'intero PC e ha lo scopo di eseguire i programmi e gestire gli altri componenti del computer.
Al giorno d'oggi è possibile avere un computer con più di una CPU, si parla in questo caso di *multicore*. Un computer multicore è in grado di supportare il calcolo parallelo al livello di architettura. Quindi questo tipo di programmazione può essere visto come il processo di inviare a ciascun core una parte di calcolo. I core lavorano contemporaneamente, ma soprattutto indipendentemente l'uno dall'altro, elaborando dei risultati che, messi insieme in un secondo momento, andranno a costruire il risultato globale del problema di partenza.

4.2 Programmazione in parallelo con CUDA

Oltre alla CPU esiste anche un altro microprocessore, chiamato GPU (graphics processing unit). Esso si contraddistingue per essere specializzato nel rendering di immagini grafiche, da cui il nome. Solo recentemente però si è pensato di far svolgere alle GPU compiti potenti e più generali, cioè non solo di tipo grafico. Inizialmente esistevano solo sistemi *omogenei*, ovvero che possono utilizzare uno o più processori necessariamente della stessa tipologia di architettura per eseguire un determinato calcolo. Pietra miliare del calcolo ad elevate prestazioni è stato l'arrivo di sistemi *eterogenei*, capaci di gestire in maniera concomitante processori con diversa architettura.

Oggigiorno, un tipico sistema eterogeneo è costituito da due CPU e due o più GPU. Una GPU opera necessariamente insieme ad una CPU, questo è il motivo per cui la prima è chiamata *device* e la seconda è chiamata *host*. Un programma eterogeneo consiste di due parti distinte:

- Host code, eseguito sulle CPU,

- Device code, eseguito sulle GPU.

Per spiegare ciò che contraddistingue CPU e GPU è necessario dare la definizione di *thread*. Un thread è una suddivisione di un processo in due o più filoni o sottoprocessi, che vengono eseguiti concorrentemente da un sistema di elaborazione. Durante l'esecuzione di un processo, vengono creati migliaia di thread. Se per eseguirne uno è necessario che ne vengano eseguiti prima altri, la CPU "aspetta", a differenza della GPU che invece passa ad eseguirne un altro. Inoltre i core della CPU sono stati costruiti in modo da ottimizzare l'esecuzione di programmi sequenziali, quelli della GPU si focalizzano sull'esecuzione di programmi in parallelo. Per supportare l'uso combinato di CPU e GPU, NVIDIA ha disegnato un modello di programmazione chiamato CUDA. La piattaforma CUDA è accesibile tramite librerie CUDA oppure estensioni di linguaggi di programmazione tra cui *C*, *C++*, *Python*, *Fortran*. I programmi sviluppati per questa tesi sono scritti tutti in CUDA $C++$.

4.3 Performance a confronto

In questo paragrafo mostreremo due codici, il primo scritto in semplice *C++*, il secondo in CUDA *C++*, e metteremo a confronto i tempi di esecuzione per dare una prima dimostrazione del fatto che il calcolo in parallelo raggiunge delle performance nettamente più elevate.
L'algoritmo implementato genera N serie distinte di 1000 numeri casuali con distribuzione normale ed è già stato presentato nel capitolo 3. Il processo-

re della macchina su cui abbiamo lavorato è *Intel Core* $i7-970$ *Processor*, caratterizzato da:

- 6 core (numero di unità di elaborazione centrale indipendenti in un solo componente di elaborazione);
- 12 thread (sequenza ordinata di base delle istruzioni che possono essere trasmesse da un'unica core CPU);
- frequenza base del processore pari a $3.20GHz$.

La GPU della macchina è la *NVIDIA GeForce GTX 580*, dotata di 512 CUDA cores.
Concettualmente, il programma scritto in *C++* effettua due cicli for, uno per la generazione di 1000 numeri casuali con distribuzione uniforme, uno che permette di iterare per N volte la creazione di 1000 numeri casuali. Il programma scritto con CUDA *C++* invece fa si che la GPU elabori parallelamente N cicli distinti, ciascuno dei quali si occupa indipendentemente dall'altro di generare la sequenza cercata.

Riportiamo di seguito, prima il codice scritto in $C++$ e poi quello scritto in CUDA $C++$ per la generazione di N sequenze di 1000 numeri casuali con distribuzione normale.

Programma in $C++$

```
#include <iostream>
#include <string.h>
#include <math.h>
#include <ctime>
#include <cstdlib>
#include <fstream>
using namespace std;
#define a0  2.50662823884
#define b0  -8.47351093090
#define a1  -18.61500062529
#define b1  23.08336743743
#define a2  41.39119773534
#define b2 -21.06224101826
#define a3  -25.44106049637
#define b3  3.13082909833
#define c0  0.3374754822726147
#define c5  0.0003951896511919
#define c1  0.9761690190917186
#define c6  0.0000321767881768
#define c2  0.1607979714918209
#define c7  0.0000002888167364
#define c3  0.0276438810333863
#define c8  0.0000003960315187
#define c4  0.0038405729373609
#define m1 2147483647
#define m2 2145483479
#define a12 63308
#define a13 .183326
#define a21 86098
#define a23 .539608
#define q12 33921
#define q13 11714
#define q21 24919
#define q23 3976
#define r12 12979
#define r13 2883
#define r21 7417
#define r23 2071
#define Invmp1 0.00000000046566128873077393
int x10=7, x11=8, x12=9, x20=10, x21=11, x22=12;
int Random(){
int h,p12,p13,p21,p23,i;
h=x10/q13;p13=-a13*(x10-h*q13)-h*r13;
h=x11/q12;p12=a12*(x11-h*q12)-h*r12;
```

```
45  if(p13<0)p13=p13+m1; if(p12<0)p12=p12+m1;
46  x10=x11; x11=x12; x12=p12-p13;
47  if(x12<0) x12=x12+m1;
48  h=x20/q23; p23=-a23*(x20-h*q23)-h*r23;
49  h=x22/q21; p21=a21*(x22-h*q21)-h*r21;
50  if(p23<0) p23=p23+m2;
51  if(p21<0) p21=p21+m2;
52  if(x12<x22) return (x12-x22+m1);
53  else return (x12-x22);}
54  double Uniform01(){
55  int Z;Z=Random(); if(Z==0) Z=m1; return (Z*Invmp1);}
56  double Inversa( double u){
57  double y,x,r; bool a=0;
58  if (u<0.5) {u=1-u; a=1; }
59  y=u-0.5;
60  if ((u)<=0.92 && u>=0.5){
61          r=y*y;
62          x = y*(((a3*r+a2)*r+a1)*r+a0)/((((b3*r+b2)*rb1)*r+b0)*r +1);}
63  else{
64          r=log(-log(1-u));
65          x=c0+r*(c1+r*(c2+r*(c3+r*(c4+r*(c5+r*(c6+r*(c7+r*c8)))))));}
66  if (a==1) x=-x;
67  return x;}
68  int main(){
69  int i,j;double u;double a;
70  for(j=0; j<1000000;j++){
71          for(i=0; i<1000; i++){
72                  u=Uniform01();
73                  a=Inversa(u);}}
74  return 0;}
```

Programma in CUDA $C++$

```
#include <iostream>
using namespace std;
#define m1 2147483647
#define m2 2145483479
#define a12 63308
#define a13 .183326
#define a21 86098
#define a23 .539608
#define q12 33921
#define q13 11714
#define q21 24919
#define q23 3976
#define r12 12979
#define r13 2883
#define r21 7417
#define r23 2071
#define Invmp1 0.00000000046566128730773937
#define a0  2.50662823884
#define b0  -8.47351093090
#define a1  -18.61500062529
#define b1  23.08336743743
#define a2  41.39119773534
#define b2  -21.06224101826
#define a3  -25.44106049637
#define b3  3.13082909833
#define c0  0.3374754822726147
#define c5  0.0003951896511919
#define c1  0.9761690190917186
#define c6  0.0000321767881768
#define c2  0.1607979714918209
#define c7  0.0000002888167364
#define c3  0.0276438810333863
#define c8  0.0000003960315187
#define c4  0.0038405729373609
#define N 1000000
#define M 2
#define THREADS_PER_BLOCK 10
__global__ void RANDOM (int*x10,int*x11,int*x12,int*x20,int*x21,int*x22,
    double*Z){
int h;int p12;int p13;int p21;int p23;
int index; index = threadIdx.x + blockIdx.x * blockDim.x;
h=x10[index]/q13;p13=-a13*(x10[index]-h*q13)-h*r13;
h=x11[index]/q12;p12=a12*(x11[index]-h*q12)-h*r12;
if(p13<0)p13 =p13+m1;if(p12<0)p12 =p12+m1;
```

```
44 x10[index] =x11[index];x11[index] =x12[index];
45 x12[index] =p12-p13;if(x12[index]<0)x12[index] =x12[index]+m1;
46 h=x20[index]/q23;p23=-a23*(x20[index]-h*q23)-h*r23;
47 h=x22[index]/q21;p21=a21*(x22[index]-h*q21)-h*r21;
48 if(p23<0)p23=p23+m2;if(p21<0)p21=p21+m2;
49 if(x12[index]<x22[index])Z[index]=(x12[index]-x22[index]+m1);
50 elseZ[index]=(x12[index]-x22[index]);if(Z[index]==0)Z[index]=m1;
51 Z[index]=Z[index]*Invmp1;double y,r;bool a=0;
52 if(Z[index]<0.5){ Z[index]=1-Z[index]; a=1;}
53 y=Z[index]-0.5;if((Z[index])<=0.92&&Z[index]>=0.5){
54 r=y*y; Z[index]=y*(((a3*r+a2)*r+a1)*r+a0)/(((((b3*r+b2)*r+b1)*r+b0)*
    r+1);}
55 else{ r=log(-log(1-Z[index])); Z[index]=c0+r*(c1+r*(c2+r*(
    c3+r*(c4+r*(c5+r*(c6+r*(c7+r*c8)))))));}
56 if(a==1)Z[index]=-Z[index];}
57 int main(){
58 int *x10, *x11, *x12,*x20, *x21, *x22;double *z ;
59 int *d_x10, *d_x11, *d_x12,*d_x20, *d_x21, *d_x22;double *d_z;
60 int size = N*sizeof(int);long int size1=N*sizeof(double);int i;int n;
61 // Alloc space for device copies
62 cudaMalloc((void **)&d_x10, size);cudaMalloc((void **)&d_x11, size);
63 cudaMalloc((void **)&d_x12, size);cudaMalloc((void **)&d_x20, size);
64 cudaMalloc((void **)&d_x21, size);cudaMalloc((void **)&d_x22, size);
65 cudaMalloc((void **)&d_z, size1);
66 // Alloc space for host copies and setup input values
67 x10 = (int *)malloc(size); x11 = (int *)malloc(size);
68 x12 = (int *)malloc(size); x20 = (int *)malloc(size);
69 x21 = (int *)malloc(size); x22 = (int *)malloc(size);
70 z = (double *)malloc(size1);
71 for(i=0; i<N; i++){
72 x10[i]=7;x11[i]=8;x12[i]=9;x20[i]=10;x21[i]=11;x22[i]=12;}
73 // Copy inputs to device
74 cudaMemcpy(d_x10,x10,size,cudaMemcpyHostToDevice);
75 cudaMemcpy(d_x11,x11,size,cudaMemcpyHostToDevice);
76 cudaMemcpy(d_x12,x12,size,cudaMemcpyHostToDevice);
77 cudaMemcpy(d_x20,x20,size,cudaMemcpyHostToDevice);
78 cudaMemcpy(d_x21,x21,size,cudaMemcpyHostToDevice);
79 cudaMemcpy(d_x22,x22,size,cudaMemcpyHostToDevice);
80 time1=time(0);n=N/THREADS_PER_BLOCK;
81 for(i=0; i<1000; i++){
82 RANDOM<<<n,THREADS_PER_BLOCK>>>(d_x10, d_x11,d_x12,d_x20,d_x21,d_x22, d_z);}
83 cudaMemcpy(z, d_z, size1, cudaMemcpyDeviceToHost);
84 //cleanup
85 free(x10); free(x11); free(x12); free(x20); free(x21); free(x22); free(z);
86 cudaFree(d_x10); cudaFree(d_x11); cudaFree(d_x12);
87 cudaFree(d_x20); cudaFree(d_x21); cudaFree(d_x22);cudaFree(d_z);
88 return 0;}
```

Eseguiamo i due codici per tre valori distinti di N e riassumiamo i risultati di performance nella seguente tabella:

N	**Performance C++**	**Performance** $CUDA$
10000	1.498 sec.	0.018 sec.
100000	15.39 sec.	0.12859 sec.
1000000	145.2 sec.	1.14894 sec.

Tabella 4.1: Confronto tempi di calcolo

Riassumiamo inoltre le performance di CUDA nelle seguenti tre tabelle, al variare di N, dove:

- memcpy DtoH indica la parte di programma in cui le variabili vengono copiate dal device all'host;
- memcpy HtoD indica la parte di programma in cui le variabili vengono copiate dall'host al device;
- RANDOM è l'unica funzione del nostro programma;
- Time e Time% indicano rispettivamente il tempo di esecuzione e la sua percentuale, di una parte di programma (indicata nell'ultima colonna Name);
- Calls è il numero di volte in cui la parte di programma viene eseguita;
- Avg è la media del tempo di esecuzione della parte di programma;
- Min e Max sono, rispettivamente, il tempo minimo e il tempo massimo di esecuzione della parte del programma;
- us sono i microsecondi, ms i millisecondi.

Time(%)	Time	Calls	Avg	Min	Max	Name
57.26%	25.194ms	1001	25.168us	25.119us	25.664us	memcpy DtoH
42.59%	18.738ms	1000	18.737us	15.748us	18.917us	RANDOM
0.15%	65.023us	6	10.837us	10.240us	13.472us	memcpy HtoD

Tabella 4.2: Tempi di calcolo CUDA, N=10000

Time(%)	Time	Calls	Avg	Min	Max	Name
64.98%	240.06ms	1001	239.82us	239.74us	240.41us	memcpy DtoH
34.81%	128.59ms	1000	128.59us	108.13us	130.09us	RANDOM
0.21%	770.96us	6	128.49us	128.03us	129.31us	memcpy HtoD

Tabella 4.3: Tempi di calcolo CUDA, $N = 100000$

Time(%)	Time	Calls	Avg	Min	Max	Name
67.56%	2.40850s	1001	2.4061ms	2.4046ms	3.4587ms	memcpy DtoH
32.23%	1.14894s	1000	1.1489ms	973.12us	1.1586ms	RANDOM
0.21%	7.6401ms	6	1.2734ms	1.2712ms	1.2764ms	memcpy HtoD

Tabella 4.4: Tempi di calcolo CUDA, $N = 1000000$

Come possiamo notare, i risultati sono sorprendenti: in ogni caso il programma in CUDA C++ impiega meno di 2 secondi per generare il risultato richiesto, a differenza del codice in C++ che impiega 145 se $N = 1000000$.
Inoltre, osserviamo dalle ultime tre tabelle, che la maggior parte del tempo di esecuzione del codice scritto in CUDA C++ viene impiegata per trasferire le informazioni dalla GPU alla CPU (più del 50% del tempo totale).
Ai fini di questa tesi, sarà necessario implementare dei programmi che prevedono centinaia di milioni generazioni di numeri casuali. Senza l'utilizzo di CUDA, i tempi sarebbero infinitamente lunghi. Ecco giustificato il motivo della scelta di lavorare con il calcolo in parallelo.

4.4 Risultati numerici

Vogliamo prima di tutto applicare il metodo Monte Carlo appena descritto, per ricavare il prezzo di uno Zero Coupon Bond.
Per far ciò è necessario fissare i valori dei parametri del modello. Poniamo:

$$\begin{array}{llll} \sigma = 0.2 & \beta = 0.5 & b = 0.1 & c = 1 \\ k = 2.0 & \theta = 0.03 & \rho = -0.3 & \delta = 0.05 \end{array}$$

Inoltre, fissiamo i valori iniziali di X e di r ponendo $X_0 = 0$ e $r_0 = 0.02$. Il numero di punti M in cui viene discretizzato l'intervallo temporale $[0, T]$ è esattamente $T \cdot 100$, ovvero 100 punti per ogni anno.
Prima di mostrare i risultati, riportiamo di seguito il codice scritto in CUDA C++.

Zero Coupon Bond in CUDA $C++$

```
1
2  #include <float.h>
3  #include <time.h>
4  #include <iostream>
5  #include <gsl/gsl_vector.h>
6  using namespace std;
7  #define m1 2147483647
8  #define m2 2145483479
9  #define a12 63308
10 #define a13 .183326
11 #define a21 86098
12 #define a23 .539608
13 #define q12 33921
14 #define q13 11714
15 #define q21 24919
16 #define q23 3976
17 #define r12 12979
18 #define r13 2883
19 #define r21 7417
20 #define r23 2071
21 #define Invmp1 0.00000000046566128730773937
22 #define a0  2.50662823884
23 #define b0  -8.47351093090
24 #define a1  -18.61500062529
25 #define b1  23.08336743743
26 #define a2  41.39119773534
27 #define b2 -21.06224101826
28 #define a3  -25.44106049637
29 #define b3  3.13082909833
30 #define c0  0.3374754822726147
31 #define c5  0.0003951896511919
32 #define c1  0.9761690190917186
33 #define c6  0.0000321767881768
34 #define c2  0.1607979714918209
35 #define c7  0.0000002888167364
36 #define c3  0.0276438810333863
37 #define c8  0.0000003960315187
38 #define c4  0.0038405729373609
39 #define N 1000000
40 #define THREADS_PER_BLOCK 1000
41 #define sigma 0.2
42 #define theta 0.03
43 #define beta 0.5
44 #define delta 0.05
```

```
45 #define c 1
46 #define k 2.0
47 #define b 0.1
48 #define X0 0.0
49 #define r0 0.02
50 #define rho -0.3
51
52 __global__ void PATH1(double *Z1,double *Z2, double *X, double *r, double
       Delta, double sqrtDelta){
53 int index;
54 index = threadIdx.x + blockIdx.x * blockDim.x;
55 X[index]=X[index]+(r[index]+b+(c-0.5)*sigma*sigma*exp(2*(beta-1)*X[index]))
       *(Delta)
56 +sigma*exp((beta-1)*X[index])*sqrtDelta*Z1[index];
57 r[index]=r[index]+k*(theta-r[index])*Delta+delta*sqrtDelta*(rho*Z1[index]+(
       sqrt(1-rho*rho))*Z2[index]);}
58 __global__ void INTEGRALE(double *r, double *X, double *I,int m , double M){
59 double a; int index;index = threadIdx.x + blockIdx.x * blockDim.x;
60 a=b+c*sigma*sigma*exp(2*(beta-1)*X[index])+ r[index];
61 if(fabs(a)<DBL_MAX){
62 if ((m==0 )|| (m==(M-1))) I[index]=I[index]+a;
63 else if ((m\%2)==1) I[index]=I[index]+4*a;
64 else if ((m\%2)==0) I[index]=I[index]+2*a; }}
65 void prezzo(double T, double M){
66 int *x10, *x11, *x12,*x20, *x21, *x22;int *x10_1, *x11_1, *x12_1,*x20_1, *
       x21_1, *x22_1;
67 double  *z ;double Delta= T/M;double sqrtDelta=sqrt(Delta);double *r;double
       *X;double *I;
68 int *d_x10, *d_x11, *d_x12,*d_x20, *d_x21, *d_x22;
69 int *d_x10_1, *d_x11_1, *d_x12_1,*d_x20_1, *d_x21_1, *d_x22_1;
70 double *d_z_1;double *d_z;double *d_r;double *d_X;double *d_I;int size = N*
       sizeof(int);
71 long int size1=N*sizeof(double);int i;int n;
72 cudaMalloc((void **)&d_x10, size);cudaMalloc((void **)&d_x11, size);
73 cudaMalloc((void **)&d_x12, size);cudaMalloc((void **)&d_x20, size);
74 cudaMalloc((void **)&d_x21, size);cudaMalloc((void **)&d_x22, size);
75 cudaMalloc((void **)&d_z, size1);cudaMalloc((void **)&d_r, size1);
76 cudaMalloc((void **)&d_x10_1, size);cudaMalloc((void **)&d_x11_1, size);
77 cudaMalloc((void **)&d_x12_1, size);cudaMalloc((void **)&d_x20_1, size);
78 cudaMalloc((void **)&d_x21_1, size);cudaMalloc((void **)&d_x22_1, size);
79 cudaMalloc((void **)&d_z_1, size1);cudaMalloc((void **)&d_X, size1);
80 cudaMalloc((void **)&d_I, size1);
81 x10 = (int *)malloc(size); x11 = (int *)malloc(size); x12 = (int *)malloc(
       size);
82 x20 = (int *)malloc(size); x21 = (int *)malloc(size); x22 = (int *)malloc(
       size);
83 z = (double *)malloc(size1);r = (double *)malloc(size1);x10_1 = (int *)
```

```
        malloc(size);
84  x11_1 = (int *)malloc(size); x12_1 = (int *)malloc(size);x20_1 = (int *)
        malloc(size);
85  x21_1 = (int *)malloc(size); x22_1 = (int *)malloc(size);X = (double *)
        malloc(size1);
86  I = (double *)malloc(size1);
87  srand(time(0));
88  for(i=0; i<N; i++){
89  x10[i]=rand()%1000+1; x11[i]=rand()%1000+1;
90  x12[i]=rand()%1000+1;x20[i]=rand()%1000+1;
91  x21[i]=rand()%1000+1;x22[i]=rand()%1000+1;
92  x10_1[i]=rand()%1000+1;x11_1[i]=rand()%1000+1;
93  x12_1[i]=rand()%1000+1;x20_1[i]=rand()%1000+1;
94  x21_1[i]=rand()%1000+1;x22_1[i]=rand()%1000+1;
95  r[i]=r0;X[i]=X0;I[i]=0;}
96  cudaMemcpy(d_x10, x10, size, cudaMemcpyHostToDevice);
97  cudaMemcpy(d_x11, x11, size, cudaMemcpyHostToDevice);
98  cudaMemcpy(d_x12, x12, size, cudaMemcpyHostToDevice);
99  cudaMemcpy(d_x20, x20, size, cudaMemcpyHostToDevice);
100 cudaMemcpy(d_x21, x21, size, cudaMemcpyHostToDevice);
101 cudaMemcpy(d_x22, x22, size, cudaMemcpyHostToDevice);
102 cudaMemcpy(d_r, r, size1, cudaMemcpyHostToDevice);
103 cudaMemcpy(d_x10_1, x10_1, size, cudaMemcpyHostToDevice);
104 cudaMemcpy(d_x11_1, x11_1, size, cudaMemcpyHostToDevice);
105 cudaMemcpy(d_x12_1, x12_1, size, cudaMemcpyHostToDevice);
106 cudaMemcpy(d_x20_1, x20_1, size, cudaMemcpyHostToDevice);
107 cudaMemcpy(d_x21_1, x21_1, size, cudaMemcpyHostToDevice);
108 cudaMemcpy(d_x22_1, x22_1, size, cudaMemcpyHostToDevice);
109 cudaMemcpy(d_X, X, size1, cudaMemcpyHostToDevice);
110 cudaMemcpy(d_I, I, size1, cudaMemcpyHostToDevice);
111 n=N/THREADS_PER_BLOCK;
112 for(i=0 ; i<M; i++){
113 INTEGRALE<<<n,THREADS_PER_BLOCK>>>(d_r, d_X,d_I,i,M);
114 RANDOM<<<n,THREADS_PER_BLOCK>>>(d_x10, d_x11,d_x12,d_x20,d_x21,d_x22, d_z);
115 RANDOM<<<n,THREADS_PER_BLOCK>>>(d_x10_1, d_x11_1,d_x12_1,d_x20_1,d_x21_1,
        d_x22_1, d_z_1);
116 PATH1<<<n,THREADS_PER_BLOCK>>>(d_z_1, d_z,d_X, d_r, Delta, sqrtDelta);
117 cudaMemcpy(I,d_I, size1, cudaMemcpyDeviceToHost);}
118 cudaMemcpy(I,d_I, size1, cudaMemcpyDeviceToHost);
119 double PRICE=0;double confidence;
120 for(i=0; i<N; i++){  I[i]=I[i]*Delta/3; PRICE=PRICE + exp(-I[i]); }
121 PRICE=PRICE/N;
122 for(i=0; i<N; i++){  confidence=confidence+ (PRICE-exp(-I[i]))*(PRICE-exp(-I
        [i])); }
123 confidence=confidence/(N-1);confidence=sqrt(confidence);
124 free(x10); free(x11); free(x12); free(x20); free(x21); free(x22); free(z);
125 cudaFree(d_x10); cudaFree(d_x11); cudaFree(d_x12);
```

```
126 cudaFree(d_x20); cudaFree(d_x21); cudaFree(d_x22); cudaFree(d_z);}
127 int main(){
128 clock_t start,end;
129 for(int i=1; i<17; i++){
130 start=clock();prezzo(0.25*i, 25*i);end=clock();
131 cout << ((double)(end-start))/CLOCKS_PER_SEC <<endl;}
132 return 0;}
```

Nelle seguenti tre tabelle vengono mostrati i risultati al variare del tempo di scadenza T. Nella prima tabella abbiamo usato un numero di simulazioni N pari a 10000, nella seconda pari a 100000 e nella terza pari a 1000000.
La colonna **Confidence** indica una stima degli estremi dell'intervallo di confidenza, la colonne **Tempo CUDA** e **Tempo C++** indicano, rispettivamente, il tempo che hanno impiegato il codice scritto in CUDA C++ e quello scritto in C++ a generare il risultato.

T	**Prezzo**	**Confidence**	**Tempo CUDA**	**Tempo C++**
0.25	0.964291	$7.16e-05\pm$	0.006689 sec.	0.157 sec.
0.5	0.927252	0.000180989±	0.009414 sec.	0.201 sec.
0.75	0.889671	0.000286496±	0.012456 sec.	0.313 sec.
1	0.854126	0.000375805±	0.014853 sec.	0.37 sec.
1.25	0.818959	0.000454127±	0.017457 sec.	0.457 sec.
1.5	0.786132	0.000518999±	0.020514 sec.	0.542 sec.
1.75	0.753859	0.00057587±	0.023039 sec.	0.615 sec.
2	0.723841	0.000622268±	0.025324 sec.	0.72 sec.
2.25	0.694505	0.000662561±	0.028425 sec.	0.799 sec.
2.5	0.667247	0.000694807±	0.030857 sec.	0.888 sec.
2.75	0.640503	0.000721754±	0.033454 sec.	0.976 sec.
3	0.615625	0.000744696±	0.036592 sec.	1.081 sec.
3.25	0.591233	0.000764418±	0.038893 sec.	1.197 sec.
3.5	0.568574	0.000778833±	0.041694 sec.	1.359 sec.
3.75	0.5463	0.000790726±	0.044567 sec.	1.361 sec.
4	0.525553	0.000799391±	0.047243 sec.	1.457 sec.

Tabella 4.5: Prezzo Zero Coupon Bond, N=10000

T	Prezzo	Confidence	Tempo CUDA	Tempo C++
0.25	0.964305	$2.27e-05\pm$	0.059198 sec.	1.166 sec.
0.5	0.927301	$5.63e-05\pm$	0.077743 sec.	1.754 sec.
0.75	0.889748	$8.94e-05\pm$	0.095695 sec.	2.614 sec.
1	0.8542	0.000117778±	0.115944 sec.	3.513 sec.
1.25	0.819024	0.000142691±	0.134799 sec.	4.626 sec.
1.5	0.786183	0.000163114±	0.153735 sec.	5.421 sec.
1.75	0.753916	0.000180833±	0.171076 sec.	6.238 sec.
2	0.723922	0.000195388±	0.192115 sec.	7.237 sec.
2.25	0.694615	0.000209021±	0.210947 sec.	8.251 sec.
2.5	0.667279	0.00021908±	0.229986 sec.	9.12 sec.
2.75	0.640491	0.000227697±	0.244314 sec.	9.94 sec.
3	0.61559	0.000234806±	0.263891 sec.	11.143 sec.
3.25	0.591162	0.000245111±	0.27981 sec.	11.88 sec.
3.5	0.56845	0.000250423±	0.299232 sec.	12.967 sec.
3.75	0.546176	0.000255066±	0.31473 sec.	13.844 sec.
4	0.525449	0.000258919±	0.336202 sec.	14.706 sec.

Tabella 4.6: Prezzo Zero Coupon Bond, N=100000

T	Prezzo	Confidence	Tempo CUDA	Tempo C++
0.25	0.964298	$7.195e-05\pm$	0.513402 sec.	9.24 sec.
0.5	0.927273	$1.792e-05\pm$	0.644397 sec.	18.081 sec.
0.75	0.889753	$2.837e-05\pm$	0.771951 sec.	27.125 sec.
1	0.854211	$3.740e-05\pm$	0.921153 sec.	35.96 sec.
1.25	0.819061	$4.532e-05\pm$	1.081400 sec.	45.509 sec.
1.5	0.786222	$5.188e-05\pm$	1.217760 sec.	54.159 sec.
1.75	0.753915	$5.755e-05\pm$	1.343030 sec.	64.815 sec.
2	0.723908	$6.230e-05\pm$	1.486160 sec.	74.934 sec.
2.25	0.69454	$6.622e-05\pm$	1.630040 sec.	79.743 sec.
2.5	0.667218	$6.958e-05\pm$	1.773100 sec.	88.288 sec.
2.75	0.640447	$7.271e-05\pm$	1.915630 sec.	98.35 sec.
3	0.615548	$7.504e-05\pm$	2.057940 sec.	106.376 sec.
3.25	0.591113	$7.712e-05\pm$	2.203580 sec.	115.646 sec.
3.5	0.568459	$7.838e-05\pm$	2.347230 sec.	124.434 sec.
3.75	0.54613	$8.023e-05\pm$	2.490630 sec.	138.067 sec.
4	0.525408	$8.070e-05\pm$	2.630870 sec.	145.066 sec.

Tabella 4.7: Prezzo Zero Coupon Bond, N=1000000

Come ci aspettavamo, c'è una differenza sostanziale tra i tempi di esecuzione dei due codici, che si rivelerà fondamentale per il problema che presentiamo nel prossimo capitolo.

Capitolo 5

Calibrazione

I risultati ottenuti fino ad ora necessitano di tempi di calcolo non esageratamente lunghi, quindi l'utilizzo di CUDA al momento non è indispensabile. Se parliamo di calibrazione di un modello finanziario però, la situazione cambia.
La calibrazione di un modello finanziario consiste nel ricavare i parametri del modello in modo che, i prezzi calcolati con questo, si avvicinino il più possibile ai reali prezzi di mercato. L'assunzione su cui si fonda è la presenza di un numero sufficiente di opzioni liquide trattate sul mercato, per le quali è possibile valutare il prezzo in modo rapido ed efficiente. Nella calibrazione si vuole quindi ricavare i parametri che descrivono una dinamica risk-neutral per i prezzi osservati sul mercato. La calibrazione è quindi il problema inverso associato a quello della valutazione delle opzioni. In generale però, la soluzione del problema di calibrazione, se esiste, non è detto che sia unica. In ogni caso si cercano i parametri che consentono la migliore approssimazione dei prezzi di mercato all'interno di una data classe di modelli.
Il problema di calibrazione si potrebbe gestire come un problema di ottimizzazione. In particolar modo, in questa tesi si è scelto di minimizzare l'errore relativo medio, cercando:

$$\min_{x\in\Omega} \frac{1}{m} \sum_{i=1}^{m} \frac{(p(x) - p_i)^2}{p_i^2}, \tag{5.1}$$

dove

- x è il vettore dei parametri che vogliamo calibrare;
- $p(x)$ indica il prezzo calcolato con il nostro modello;
- m è il numero totale di prezzi di mercato su cui facciamo la calibrazione;
- p_i è il prezzo di mercato;
- Ω indica l'insieme in cui cerchiamo i parametri da calibrare.

Molti problemi di ottimizzazione derivanti da applicazioni reali sono caratterizzati dal fatto che l'espressione analitica della funzione obiettivo non è nota. Tale situazione si verifica anche nel nostro problema. É chiaro che in questo caso non è possibile (o comunque richiede un costo troppo elevato) calcolare le derivate, di solito fondamentali nel calcolo del minimo di una funzione. L'interesse applicativo ha motivato quindi lo sviluppo di metodi di ottimizzazione che non richiedano la conoscenza delle derivate, tra cui l'algoritmo Nelder-Mead. Il metodo Nelder-Mead, noto anche come metodo del simplesso, utilizza una tecnica di ricerca euristica del minimo di una funzione, basandosi sul concetto di simplesso.
Esso è costituito da sei passi, che vengono ripetuti iterativamente, finché non si raggiunge una condizione di convergenza. Prima di tutto però è necessario scegliere un simplesso iniziale n-dimensionale (dove n è esattamente il numero di parametri che si vogliono calibrare). Indichiamo con f la funzione da minimizzare, che nel nostro caso sarà $f(x) := \frac{1}{m}\sum_{i=1}^{m}\frac{(p(x)-p_i)^2}{p_i^2}$. Descriviamo di seguito i passi:

1. prima di tutto si ordinano i vertici del simplesso, $x_1, \ldots, x_{n+1}$ in modo tale che
$$f(x_1) < f(x_1) < \ldots f(x_{n+1});$$
2. si calcola il baricentro x_0 dei vertici $x_1, \ldots, x_n$;

3. si calcola il punto riflesso $x_r := x_0 + \alpha(x_0 - x_{n+1})$. Se x_r è tale che $f(x_1) \leq f(x_r) < f(x_n)$, allora si sostituisce x_{n+1} con x_r e si ritorna al punto 1;

4. se il punto x_r è tale che $f(x_r) < f(x_1)$, si calcola il punto espanso $x_e := x_0 + \gamma(x_0 - x_{n+1})$. Se x_e è tale che $f(x_e) < f(x_r)$, si sostituisce x_{n+1} con x_e e si ritorna al punto 1. Altrimenti si sostituisce x_{n+1} con x_r e si torna al punto 1. Nel caso restante in cui il punto riflesso x_r è tale che $f(x_r) \geq f(x_n)$, si passa al punto successivo;

5. si determina il punto di contrazione $x_c := x_0 + \delta(x_0 - x_{n+1})$. Se $f(x_c) < f(x_{n+1})$, si sostituisce x_{n+1} con x_c e si torna al punto 1, altrimenti si passa al punto successivo;

6. tutti i punti tranne x_0 vengono sostituiti con $x_i := x_1 + \sigma(x_i - x_1)$.

I coefficienti α, β, γ e σ sono i coefficienti di riflessione, espansione, contrazione e riduzione e comunemente si sceglie $\alpha = 1$, $\gamma = 2$, $\delta = -\frac{1}{2}$ e $\sigma = \frac{1}{2}$.
La determinazione del simplesso iniziale è importante, in quanto un simplesso iniziale troppo piccolo può portare ad una ricerca locale, aumentando il rischio di insuccesso. Per tale motivo, il simplesso iniziale deve essere costruito accuratamente e facendo attenzione alla natura del problema.
La libreria *gsl* di $C++$ ha in sé delle funzioni che permettono di risolvere il problema di ottimizzazione definito dalla (5.1) applicando lalgoritmo di Nelder-Mead.
Il nostro obiettivo è quello di calibrare il modello descritto dalla (2.7) utilizzando 40 valori di mercato, con $T = 0.25, 0.5, 0.75, 1, \ldots, 3.75, 4$. Osserviamo che per calcolare il valore di $f(x)$, è necessario calcolare 40 valori attesi al variare di T (ho una sommatoria di 40 termini). In più, l'algoritmo di Nelder e Mead prevede che la funzione f venga calcolata un numero elevato di volte, che risulteranno essere circa 100. Ne risulta che per calibrare il modello, bisogna calcolare circa 4000 valori attesi. Nel terzo capitolo abbiamo visto che in media, utilizzando 100000 simulazioni, un valore atteso viene calcolato da un codice in CUDA C++ in circa 0.35 secondi, da un codice in C++ in

circa 15 secondi. Di conseguenza, la calibrazione in C++ verrebbe eseguita in $15 \cdot 4000$ secondi ovvero più di 16 ore, a differenza della calibrazione in CUDA C++ che impiega $0.35 \cdot 4000$ secondi, ovvero circa 23 minuti. Calibrare il nostro modello senza ricorrere al calcolo in parallelo risulta quindi impensabile.
Prima di calibrare il nostro modello, introduciamo i Credi Default Swaps.

5.1 Credi Default Swaps

Il Credit Default Swap (CDS) è un contratto appartenente alla categoria dei derivati sul rischio di credito che offre la possibilità di coprirsi dall'eventuale insolvenza di un debitore contro il pagamento di un premio periodico. Più precisamente, è un contratto con il quale il *protection buyer* si impegna a pagare una somma fissa periodica fino ad un tempo prestabilito T, a favore della controparte, il *protection seller* che si assume il rischio di credito gravante su quella attività nel caso in cui si verifichi un evento di default futuro, cioè nel caso in cui $\tau \in [0, T]$.
La somma periodica che il creditore paga è in genere commisurata al rischio e alla probabilità di insolvenza del soggetto terzo debitore. L'aspetto fondamentale del CDS consiste nel fatto che sia il protection buyer che il protection seller, possono anche non avere alcun rapporto di credito con il terzo soggetto, in quanto il contratto prescinde dalla presenza di quest'ultimo; il sottostante è unicamente il merito creditizio e non il vero e proprio credito.
La somma R che il protection buyer deve pagare in tempi prestabiliti $T_1, \ldots, Tm \in [0, T]$ se non avviene il default, è chiamata premio alla pari. Assumiamo che $T_j - T_i$ sia costante per ogni $i \neq j$ e lo denotiamo con α.
Il premio alla pari viene calcolato in modo tale che il contratto abbia un valore attuale pari a zero a scadenza. Ciò perché il valore atteso del pagamento del venditore è esattamente uguale e opposto al valore atteso del premio pagato dall'acquirente. Chiamiamo pagamento di default il pagamento che il protection seller è obbligato a pagare al protection buyer se avviene il default.

Consideriamo il pagamento di default come una variabile Z_τ, che è diversa da zero se $\tau \in [0,T]$. Riprendendo il modello definito dalla (2.7) e denotando con $D(t,s) := e^{-\int_t^s r_u du}$ il fattore di sconto stocastico, il payoff di un CDS al tempo t è dato dalla seguente equazione:

$$CSDH_t(R) = \mathbb{1}_{\{t<\tau\leq T\}} Z_\tau D(t,\tau) - \sum_{i=1}^{M} \alpha R \mathbb{1}_{\{\tau>T_i\}} D(t,T_i). \tag{5.2}$$

Quindi il prezzo è:

$$CDSP_t(R) = E\left[\mathbb{1}_{\{t<\tau\leq T\}} Z_\tau D(t,\tau) - \sum_{i=1}^{M} \alpha R \mathbb{1}_{\{\tau>T_i\}} D(t,T_i) | \mathscr{G}_t\right]. \tag{5.3}$$

Per trovare una formula esplicita che ci permetta di calcolare il valore di $R(t,T)$, e poter calibrare il modello, assumiamo che il pagamento di default Z_τ abbia valore costante L.
Dal lemma (1.1.3), abbiamo che:

$$CDSP_t(R) = L\mathbb{1}_{\{\tau>t\}} E[\int_t^T D(t,v) e^{\Gamma_t - \Gamma_v} d\Gamma_v | \mathscr{F}_t] - \alpha R \sum_{i=1}^{M} \mathbb{1}_{\{\tau>T_i\}} E[D(t,T_i) e^{\Gamma_t - \Gamma_{T_i}} | \mathscr{F}_t] =$$

$$= D^{-1}(0,t) e^{\Gamma_t} \mathbb{1}_{\{\tau>t\}} \left(LE\left[\int_t^T D(0,v) e^{-\Gamma_v} d\Gamma_v | \mathscr{F}_t\right] - \alpha R \sum_{i=1}^{M} E[D(0,T_i) e^{-\Gamma_{T_i}} | \mathscr{F}_t] \right) =$$

$$= e^{\int_0^t (r_s+\gamma_s) ds} \mathbb{1}_{\{\tau>t\}} \left(LE\left[\int_t^T e^{-\int_0^v (r_s+\gamma_s) ds} \gamma_v dv | \mathscr{F}_t\right] - \alpha R \sum_{i=1}^{M} E[e^{-\int_0^{T_i} (r_s+\gamma_s) ds} | \mathscr{F}_t] \right) =$$

$$= e^{\int_0^t (r_s+\gamma_s) ds} \mathbb{1}_{\{\tau>t\}} \left(L \int_t^T E[e^{-\int_0^v (r_s+\gamma_s) ds} \gamma_v | \mathscr{F}_t] dv - \alpha R \sum_{i=1}^{M} E[e^{-\int_0^{T_i} (r_s+\gamma_s) ds} | \mathscr{F}_t] \right).$$

Infine, ponendo $CDSP_t(R)$ uguale a zero, possiamo esplicitare il valore di $R(t,T)$, e cioè:

$$R(t,T) = \frac{L \int_t^T E[e^{-\int_0^v (r_s+\gamma_s) ds} \gamma_v | \mathscr{F}_t]}{\alpha \sum_{i=1}^{M} E[e^{-\int_0^{T_i} (r_s+\gamma_s) ds} | \mathscr{F}_t]} \tag{5.4}$$

Osservazione 1. Assumendo che il coefficiente di correlazione ρ dei moti Browniani sia nullo, risulta:

$$CSDP_t(R) = e^{\int_0^t (r_s+\gamma_s) ds} \mathbb{1}_{\{\tau>t\}} (L \int_t^T E[e^{-\int_0^v r_s ds} | \mathscr{F}_t] E[e^{-\int_0^v \gamma_s ds} \gamma_v | \mathscr{F}_t] dv -$$

$$\alpha R \sum_{i=1}^{M} \mathbb{1}_{\{\tau>t\}} E[e^{-\int_0^{T_i} r_s ds}|\mathscr{F}_t] E[e^{-\int_0^{T_i} \gamma_s ds}|\mathscr{F}_t])$$

Ricordiamo che $Q(t,T) = E[e^{-\int_0^T \gamma_s ds}|\mathscr{F}_t]$ è la survival probability, per cui ponendo $p(t,x,r,T) = E[e^{-\int_0^T r_s ds}|\mathscr{F}_t]$, la formula dei CDS spread possiamo riscriverla come:

$$R(t,T) = \frac{L\int_t^T p(t,x,r,v)\frac{\partial Q}{\partial v}(t,v)dv}{\alpha\sum_{i=1}^{M} p(t,x,r,T_i)Q(t,T_i)}. \tag{5.5}$$

Dalla (5.5) possiamo ricavare una formula per calcolare ricorsivamente le survival probabilities $Q(t,T)$:

$$\begin{aligned} Q(t,T_1) &= \frac{\alpha R(t,T_1)}{\alpha R(t,T_1)+L} \\ Q(t,T_i) &= \frac{-(\alpha R(t,T_i)+L)\sum_{h=1}^{i-1} D(t,T_h)Q(t,T_h) + L\sum_{h=1}^{i} D(t,T_h)Q(t,T_{h-1})}{\alpha R(t,T_i)D(t,T_i)+LD(t,T_i)}. \end{aligned} \tag{5.6}$$

Sostituendo nella (5.6) i dati di mercato, possiamo calibrare il modello JD-CEV attraverso le survival probabilities.

5.2 Risultati calibrazione

Il nostro obiettivo è quello di calibrare i coefficienti β, σ, b e c del modello JDVEC su 40 survival probabilities calcolate mediante la (5.6) con i dati di mercato, con T_i che varia da 0.25 anni a 4 anni.
Fissiamo i parametri del Vasiceck ponendo $k = 0.154777$, $\theta = 0.02305898$, $\delta = 0.009737$. Inoltre poniamo $X_0 = 0$, $r_0 = -0.038$, $\rho = 0$.
Nel codice CUDA $C++$ per la calibrazione abbiamo utilizzato la libreria gsl di $C++$ per risolvere il problema di ottimizzazione con il metodo Nelder-Mead, imponendo i seguenti vincoli sui coefficienti da calibrare: $\beta < 1$, $\sigma > 0$, $b \geq 0$ e $c \geq 0$. Il numero di simulazioni N utilizzato nel metodo Monte Carlo è 50000.

Mostriamo di seguito due tabelle contenenti le survival probabilities calcolate con il metodo Monte Carlo, dopo la calibrazione; gli stessi risultati sono stati riportati su un grafico.
I coefficienti ottenuti dalla calibrazione sono: $\sigma = 0.214134$, $\beta = 0.0482804$, $b = 0.001523$, $c = 0.0946773$.

T	Prob. modello	Prob. mercato	Err. relativo	Err. assoluto
0.25	0.998519	0.99689	0.16337%	0.16286%
0.5	0.996955	0.99752	0.05665%	0.05651%
0.75	0.995305	0.99598	0.06779%	0.06752%
1	0.993646	0.99425	0.06074%	0.06039%
1.25	0.991877	0.99263	0.07590%	0.07534%
1.5	0.990024	0.99051	0.04903%	0.04856%
1.75	0.987902	0.98794	0.00388%	0.00383%
2	0.985588	0.98508	0.05158%	0.05081%
2.25	0.982766	0.98218	0.05968%	0.05861%
2.5	0.979734	0.9792	0.05456%	0.05343%
2.75	0.976269	0.97598	0.02960%	0.02889%
3	0.972261	0.9723	0.00397%	0.00386%
3.25	0.967602	0.96797	0.03802%	0.03680%
3.5	0.963105	0.96306	0.46988%	0.00453%
3.75	0.958061	0.9578	0.02720%	0.02605%
4	0.952794	0.95254	0.02671%	0.02544%
4.25	0.947192	0.94762	0.04520%	0.04283%
4.5	0.941763	0.94291	0.12167%	0.11473%
4.75	0.936532	0.93822	0.17989%	0.16878%
5	0.931093	0.93328	0.23432%	0.21869%
5.25	0.925278	0.9279	0.28254%	0.26217%
5.5	0.919493	0.92198	0.26970%	0.24866%
5.75	0.913741	0.9156	0.20306%	0.18592%
6	0.90755	0.90887	0.14526%	0.13202%
6.25	0.901741	0.90198	0.02647%	0.02388%
6.5	0.895681	0.89495	0.08167%	0.07309%
6.75	0.889977	0.88793	0.23048%	0.20465%
7	0.884389	0.88112	0.37102%	0.32691%
7.25	0.878623	0.87471	0.44731%	0.39126%
7.5	0.872581	0.86869	0.44795%	0.38913%
7.75	0.867155	0.86304	0.47678%	0.41148%
8	0.861942	0.8578	0.48289%	0.41423%
8.25	0.856269	0.85296	0.38793%	0.33089%
8.5	0.851022	0.84846	0.30192%	0.25617%
8.75	0.846211	0.84429	0.22757%	0.19214%
9	0.840832	0.84043	0.04780%	0.04017%
9.25	0.83576	0.83683	0.12784%	0.10698%
9.5	0.83091	0.83345	0.30472%	0.25397%
9.75	0.825503	0.83025	0.57176%	0.47470%
10	0.820938	0.82716	0.75225%	0.62223%

Tabella 5.1: Survival probabilities dopo la calibrazione su 40 valori di mercato

Dopo la calibrazione, l'errore relativo medio è pari a 0.188534%, l'errore assoluto medio è pari a 0.166227%.

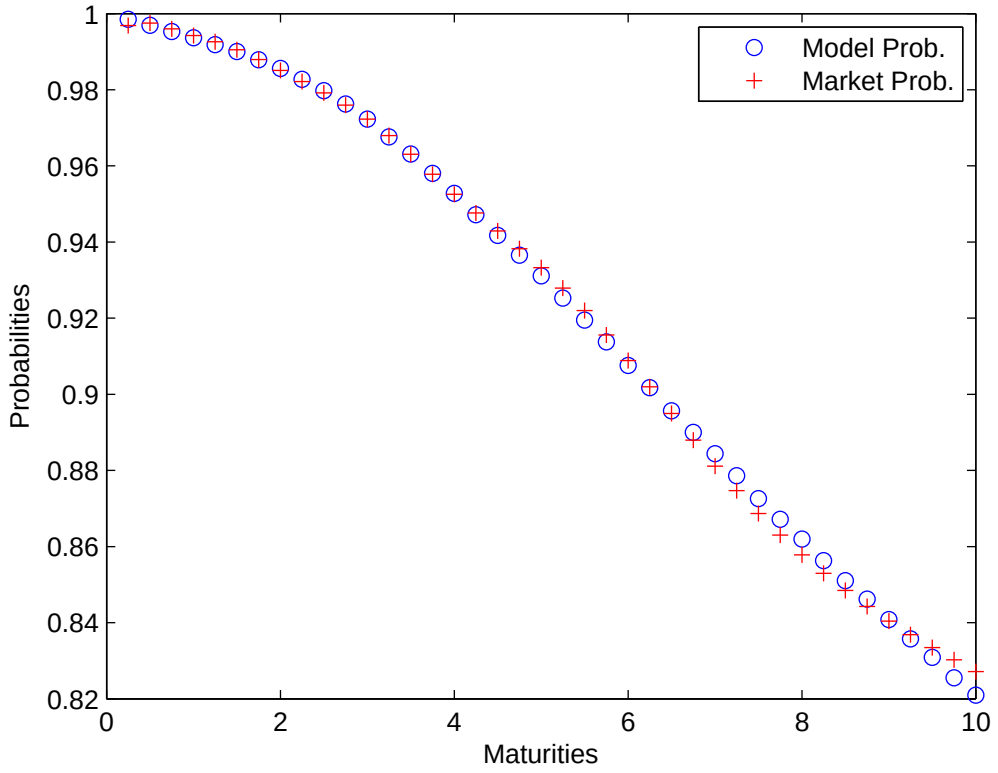

Figura 5.1: Survival probabilities dopo la calibrazione su 40 valori di mercato

Se invece di calibrare su 40 dati di mercato, calibriamo su 10 realmente quotati, otteniamo i seguenti risultati: $b = 0.00051$, $c = 0.110662$, $\beta = 0.0521187$, $\sigma = 0.215913$.
L'errore relativo medio è 0.164079%, l'errore assoluto medio è 0.149225%

T	**Prob. modello**	**Prob. mercato**	**Err. relativo**	**Err. assoluto**
0.25	0.998573	0.99689	0.16886%	0.16834%
0.5	0.99706	0.99752	0.04607%	0.04595%
0.75	0.995456	0.99598	0.05257%	0.05236%
1	0.993837	0.99425	0.04156%	0.04132%
2	0.985916	0.98508	0.08487%	0.08360%
3	0.972886	0.9723	0.06024%	0.05857%
4	0.954021	0.95254	0.15547%	0.14810%
5	0.932912	0.93328	0.03948%	0.03685%
7	0.887146	0.88112	0.68385%	0.60256%
10	0.824614	0.82716	0.30782%	0.25462%

Tabella 5.2: Survival probabilities dopo la calibrazione su 10 valori di mercato

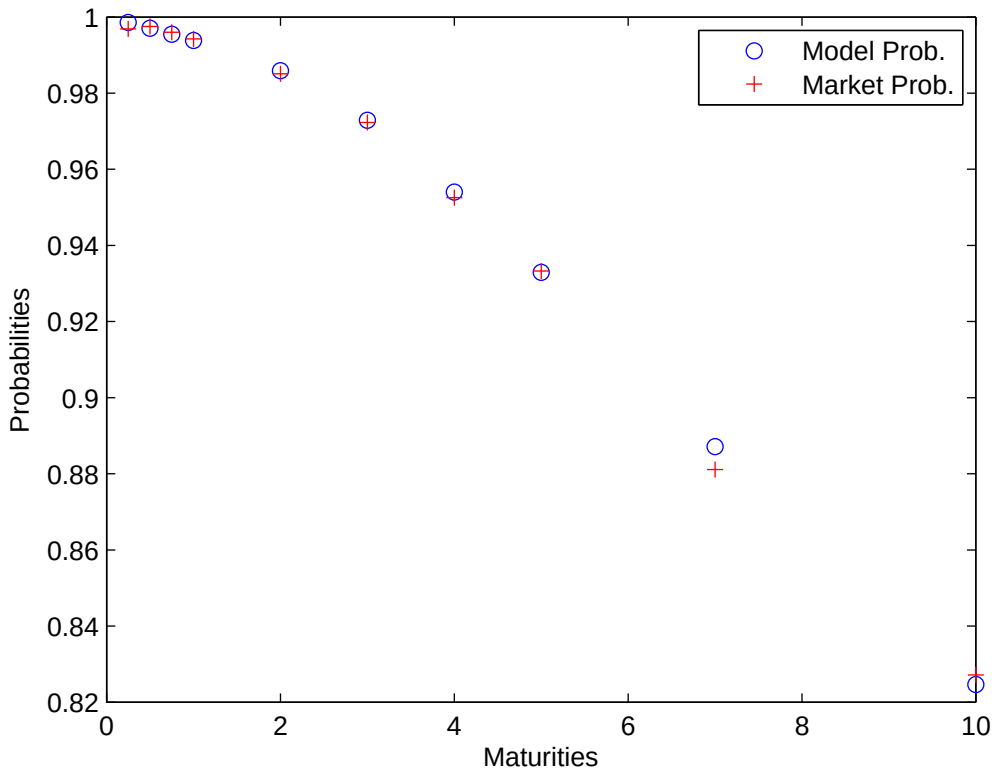

Figura 5.2: Survival probabilities dopo la calibrazione su 10 valori di mercato

Proviamo ad utilizzare i valori ottenuti calibrando il modello su 10 valori di mercato realmente quotati, per valutare le survival probabilities su tutti i 40 tempi T_i.
Otteniamo un errore relativo medio pari a 0.25264% ed un errore assoluto medio pari a 0.222269%.

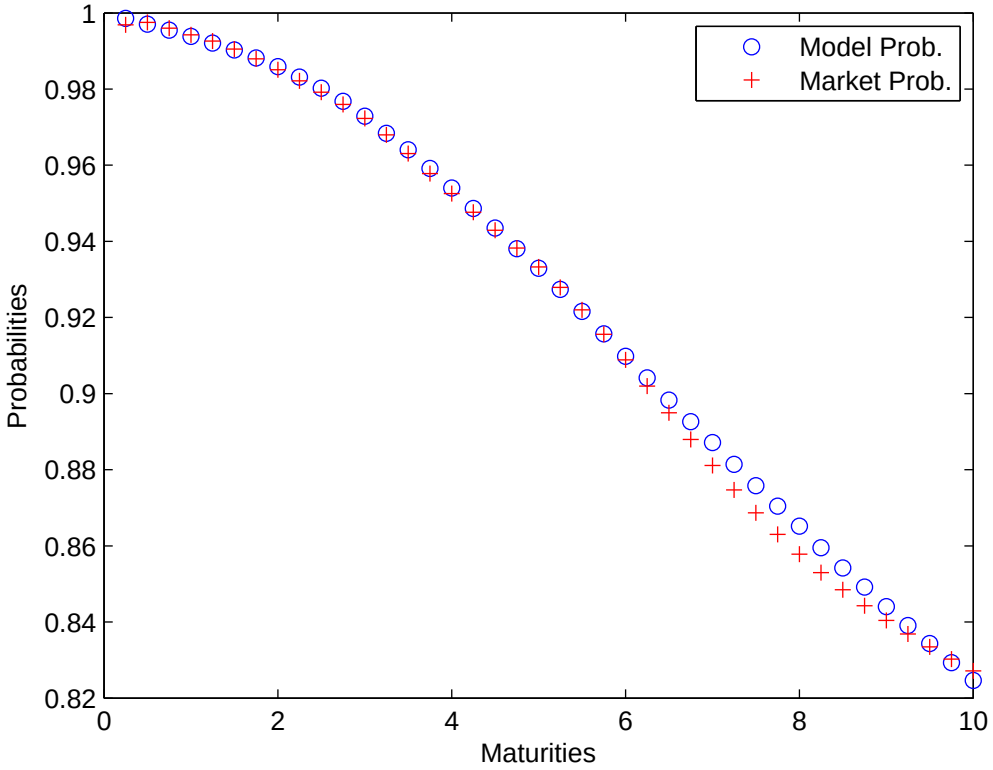

Figura 5.3: Survival probabilities valutate su 40 valori, dopo la calibrazione su 10 valori di mercato

T	Prob. modello	Prob. mercato	Err. relativo	Err. assoluto
0.25	0.998573	0.99689	0.16886%	0.16834%
0.5	0.99706	0.99752	0.04607%	0.04595%
0.75	0.995456	0.99598	0.05257%	0.05236%
1	0.993837	0.99425	0.04156%	0.04132%
1.25	0.992102	0.99263	0.05321%	0.05282%
1.5	0.990285	0.99051	0.02274%	0.02252%
1.75	0.988173	0.98794	0.02359%	0.02331%
2	0.985916	0.98508	0.08487%	0.08360%
2.25	0.983179	0.98218	0.10175%	0.09994%
2.5	0.980211	0.9792	0.10324%	0.10109%
2.75	0.976798	0.97598	0.08386%	0.08185%
3	0.972886	0.9723	0.06024%	0.05857%
3.25	0.968403	0.96797	0.04477%	0.04333%
3.5	0.964053	0.96306	0.10312%	0.09931%
3.75	0.959168	0.9578	0.14280%	0.13678%
4	0.954021	0.95254	0.15547%	0.14810%
4.25	0.948629	0.94762	0.10645%	0.10088%
4.5	0.943493	0.94291	0.06182%	0.05829%
4.75	0.938042	0.93822	0.01893%	0.01776%
5	0.932912	0.93328	0.03948%	0.03685%
5.25	0.927357	0.9279	0.05849%	0.05427%
5.5	0.921566	0.92198	0.04490%	0.04139%
5.75	0.915673	0.9156	0.00800%	0.00732%
6	0.909767	0.90887	0.09874%	0.08974%
6.25	0.904123	0.90198	0.23758%	0.21429%
6.5	0.898307	0.89495	0.37513%	0.33572%
6.75	0.892648	0.88793	0.53139%	0.47184%
7	0.887146	0.88112	0.68385%	0.60256%
7.25	0.881408	0.87471	0.76573%	0.66979%
7.5	0.875806	0.86869	0.81915%	0.71159%
7.75	0.870423	0.86304	0.85551%	0.73834%
8	0.865189	0.8578	0.86139%	0.73890%
8.25	0.859516	0.85296	0.76861%	0.65560%
8.5	0.854202	0.84846	0.67681%	0.57425%
8.75	0.849184	0.84429	0.57967%	0.48941%
9	0.844039	0.84043	0.42943%	0.36091%
9.25	0.839046	0.83683	0.26484%	0.22163%
9.5	0.834352	0.83345	0.10818%	0.09016%
9.75	0.829295	0.83025	0.11498%	0.09547%
10	0.824614	0.82716	0.30782%	0.25462%

Tabella 5.3: Survival probabilities valutate su 40 valori, dopo la calibrazione su 10 valori di mercato

Facciamo un piccola modifica al nostro modello, supponendo che i coefficienti b, c, β e γ non siano costanti in tutto l'intervallo temporale considerato $[0, 10]$, ma siano costanti a tratti, più precisamente abbiano un certo valore costante b_1, c_1, β_1 e γ_1 nella prima metà dell'intervallo $[0, 5]$, ed un valore costante diverso b_2, c_2, β_2 e γ_2 nella seconda metà dell'intervallo $[5, 10]$.
Se volessimo calibrare il modello con questa scelta sui coefficienti, il problema di ottimizzazione che prima aveva dimensione 4 poiché 4 erano i parametri da calibrare, ora avrà dimensione 8.
Mostriamo i risultati di questa nuova calibrazione, dove il numero di simulazioni N per il metodo Monte Carlo è sempre 50000.
I coefficienti ottenuti sono:
$\sigma_1 = 0.205938$, $\beta_1 = 0.0182242$, $c_1 = 0.126511$, $b_1 = 0.000589052$,
$\sigma_2 = 0.207374$, $\beta_2 = 0.048372$, $c_2 = 0.108615$, $b_2 = 0.00293748$.
L'errore relativo medio è 0.178337%, l'errore assoluto medio è 0.157845%.

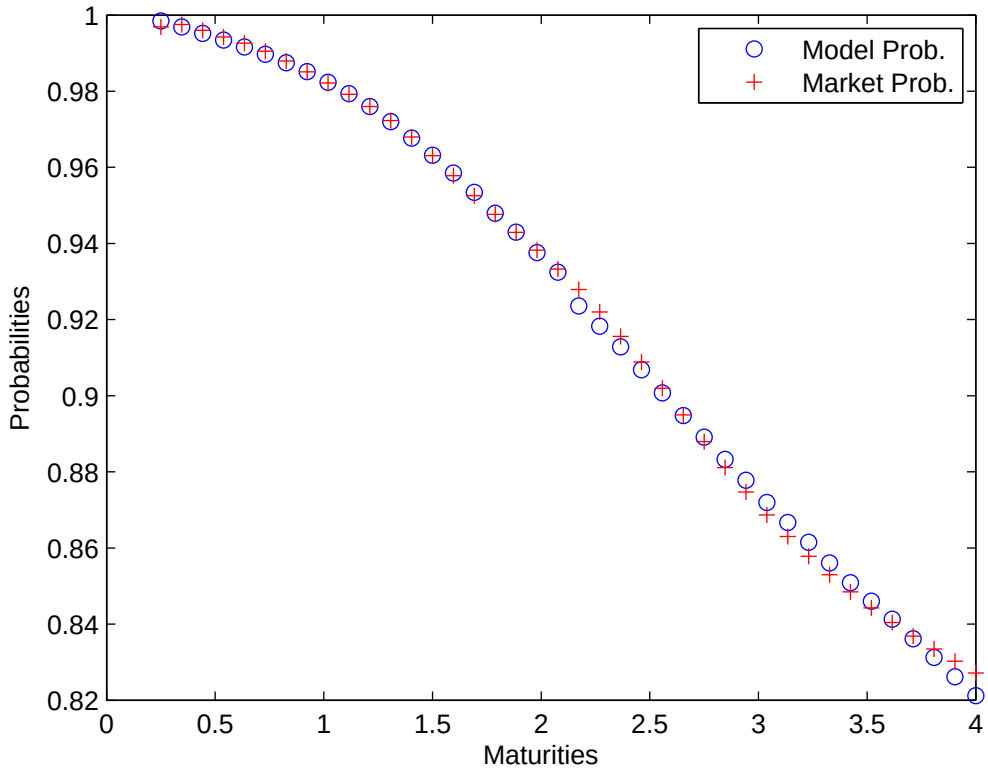

Figura 5.4: Survival probabilities con coefficienti costati a tratti, dopo la calibrazione su 40 valori di mercato

T	Prob. modello	Prob. mercato	Err. relativo	Err. assoluto
0.25	0.998481	0.99689	0.15955%	0.15906%
0.5	0.99687	0.99752	0.06521%	0.06505%
0.75	0.995162	0.99598	0.08209%	0.08176%
1	0.993439	0.99425	0.08156%	0.08110%
1.25	0.991593	0.99263	0.10445%	0.10368%
1.5	0.989675	0.99051	0.08425%	0.08345%
1.75	0.987473	0.98794	0.04726%	0.04669%
2	0.985141	0.98508	0.00614%	0.006055%
2.25	0.982319	0.98218	0.01417%	0.01392%
2.5	0.979367	0.9792	0.01704%	0.01668%
2.75	0.975966	0.97598	0.001407%	0.001373%
3	0.971971	0.9723	0.03388%	0.03294%
3.25	0.967645	0.96797	0.03362%	0.03254%
3.5	0.963178	0.96306	0.01222%	0.01177%
3.75	0.958492	0.9578	0.07229%	0.06924%
4	0.953404	0.95254	0.09070%	0.08640%
4.25	0.947931	0.94762	0.03278%	0.03106%
4.5	0.942987	0.94291	0.00812%	0.00766%
4.75	0.937572	0.93822	0.06908%	0.06482%
5	0.932434	0.93328	0.09067%	0.08462%
5.25	0.92355	0.9279	0.46880%	0.43500%
5.5	0.918259	0.92198	0.40361%	0.37212%
5.75	0.912827	0.9156	0.30290%	0.27733%
6	0.90683	0.90887	0.22442%	0.20396%
6.25	0.900735	0.90198	0.13799%	0.12447%
6.5	0.894779	0.89495	0.01912%	0.01711%
6.75	0.889133	0.88793	0.13546%	0.12028%
7	0.883257	0.88112	0.24256%	0.21372%
7.25	0.877797	0.87471	0.35291%	0.30869%
7.5	0.871976	0.86869	0.37832%	0.32864%
7.75	0.866718	0.86304	0.42617%	0.36781%
8	0.861483	0.8578	0.42931%	0.36827%
8.25	0.856048	0.85296	0.36206%	0.30882%
8.5	0.850873	0.84846	0.28444%	0.24133%
8.75	0.845995	0.84429	0.20197%	0.17052%
9	0.841273	0.84043	0.10036%	0.08434%
9.25	0.83616	0.83683	0.08001%	0.06695%
9.5	0.831235	0.83345	0.26573%	0.22147%
9.75	0.826188	0.83025	0.48921%	0.40617%
10	0.821191	0.82716	0.72164%	0.59692%

Tabella 5.4: Survival probabilities con coefficienti costati a tratti, dopo la calibrazione su 40 valori di mercato

Calibrando sui 10 valori di mercato realmente quotati otteniamo:
$\sigma_1 = 0.20379$, $\beta_1 = 0.139673$, $c_1 = 0.113236$, $b_1 = 0.00193633$,
$\sigma_2 = 0.215766$, $\beta_2 = 0.028645$, $c_2 = 0.118178$, $b_2 = 0.000273052$.
L' errore relativo medio é 0.154065%, l' errore assoluto medio é 0.142158%.

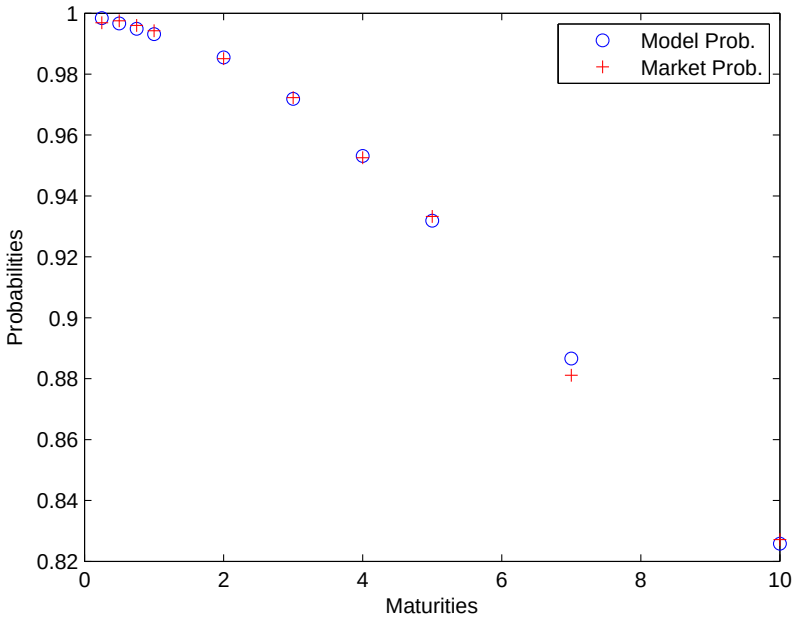

Figura 5.5: Survival probabilities con coefficienti costati a tratti, dopo la calibrazione su 10 valori di mercato

T	Prob. modello	Prob. mercato	Err. relativo	Err. assoluto
0.25	0.998371	0.99689	0.14854%	0.14808%
0.5	0.996669	0.99752	0.08535%	0.08514%
0.75	0.994891	0.99598	0.10934%	0.10890%
1	0.993129	0.99425	0.11274%	0.11209%
2	0.985514	0.98508	0.04409%	0.04343%
3	0.971889	0.9723	0.04225%	0.04108%
4	0.953111	0.95254	0.05992%	0.05707%
5	0.931892	0.93328	0.14872%	0.13879%
7	0.886635	0.88112	0.62596%	0.55155%
10	0.825806	0.82716	0.16375%	0.13545%

Tabella 5.5: Survival probabilities con coefficienti costati a tratti, dopo la calibrazione su 10 valori di mercato

I risultati sono migliorati di poco. Infatti l'errore relativo medio si è abbassato rispetto alla prima calibrazione, da 0.188534% a 0.178337%; l'errore assoluto medio si è abbassato da 0.166227% a 0.157845%.
Per quanto riguarda la calibrazione fatta su 10 valori di mercato realmente quotati, l'errore relativo medio è sceso da 0.164079% a 0.154065%, quello assoluto medio è sceso da 0.149225 a 0.142158%.
Un altro modo per effettuare una calibrazione del modello con coefficienti costanti a tratti è di spezzare il problema in due parti distinte: si calibrano separatamente i primi 4 coefficienti nell'intevallo $[0, 2]$, e gli ultimi 4 coefficienti nell'intervallo $[2, 4]$. Calibrando sui 40 valori di mercato otteniamo:
$\sigma_1 = 0.212538$, $\beta_1 = 0.0124681$, $c_1 = 0.0982962$, $b_1 = 0.00163403$
$\sigma_2 = 0.2358454$, $\beta_2 = -0.054907$, $c_2 = 0.00236352$, $b_2 = 0.00134785$
L' errore relativo medio é 0.136203%, l' errore assoluto medio é 0.120249%.

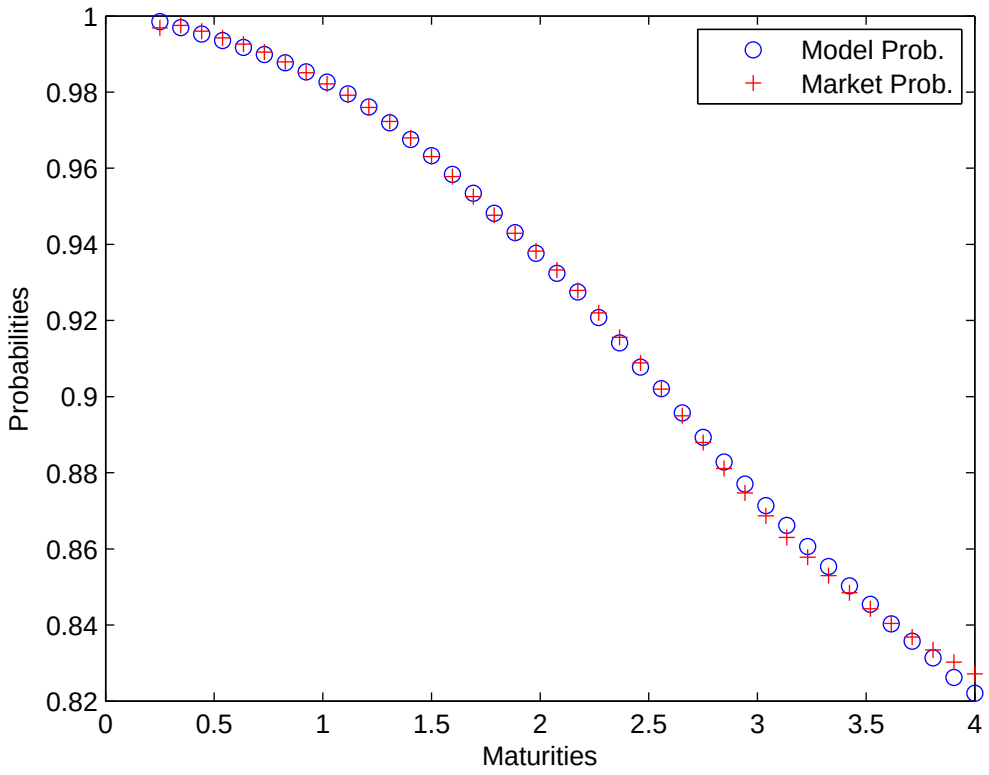

Figura 5.6: Survival probabilities con coefficienti costati a tratti, dopo la doppia calibrazione su 40 valori di mercato

T	Prob. modello	Prob. mercato	Err. relativo	Err. assoluto
0.25	0.998498	0.99689	0.16128%	0.16078%
0.5	0.996915	0.99752	0.06069%	0.06054%
0.75	0.995244	0.99598	0.07391%	0.07361%
1	0.993566	0.99425	0.06877%	0.06837%
1.25	0.991753	0.99263	0.08840%	0.08774%
1.5	0.989857	0.99051	0.06589%	0.06526%
1.75	0.987701	0.98794	0.02416%	0.02387%
2	0.985331	0.98508	0.02553%	0.02515%
2.25	0.982611	0.98218	0.04388%	0.04309%
2.5	0.979549	0.9792	0.03562%	0.03488%
2.75	0.976095	0.97598	0.01183%	0.01155%
3	0.971965	0.9723	0.03443%	0.03347%
3.25	0.967549	0.96797	0.04351%	0.04211%
3.5	0.963268	0.96306	0.02162%	0.02082%
3.75	0.958408	0.9578	0.06348%	0.06081%
4	0.953396	0.95254	0.08986%	0.08560%
4.25	0.948185	0.94762	0.05962%	0.05649%
4.5	0.943061	0.94291	0.01597%	0.01506%
4.75	0.937649	0.93822	0.06086%	0.05710%
5	0.932417	0.93328	0.09251%	0.08634%
5.25	0.92751	0.9279	0.04200%	0.03897%
5.5	0.920748	0.92198	0.13358%	0.12316%
5.75	0.914099	0.9156	0.16392%	0.15008%
6	0.907726	0.90887	0.12583%	0.11436%
6.25	0.9021	0.90198	0.01329%	0.01199%
6.5	0.895719	0.89495	0.08589%	0.07687%
6.75	0.889293	0.88793	0.15353%	0.13632%
7	0.882788	0.88112	0.18934%	0.16683%
7.25	0.877025	0.87471	0.26467%	0.23151%
7.5	0.871359	0.86869	0.30724%	0.26689%
7.75	0.86617	0.86304	0.36262%	0.31295%
8	0.86063	0.8578	0.32987%	0.28296%
8.25	0.855347	0.85296	0.27980%	0.23866%
8.5	0.850306	0.84846	0.21754%	0.18458%
8.75	0.845451	0.84429	0.13745%	0.11605%
9	0.840294	0.84043	0.01623%	0.01364%
9.25	0.835742	0.83683	0.13001%	0.10880%
9.5	0.831342	0.83345	0.25298%	0.21084%
9.75	0.826204	0.83025	0.48729%	0.40457%
10	0.822087	0.82716	0.61325%	0.50726%

Tabella 5.6: Survival probabilities con coefficienti costati a tratti, dopo la doppia calibrazione su 40 valori di mercato

Calibrando sui 10 valori di mercato otteniamo:
$\sigma_1 = 0.215353$, $\beta_1 = -0.147525$, $c_1 = 0.00415109$, $b_1 = 0.00564497$
$\sigma_2 = 0.223797$, $\beta_2 = -0.0207648$, $c_2 = 0.161838$, $b_2 = 0.00584737$
L' errore relativo medio é 0.117236%, l' errore assoluto medio é 0.109246%.

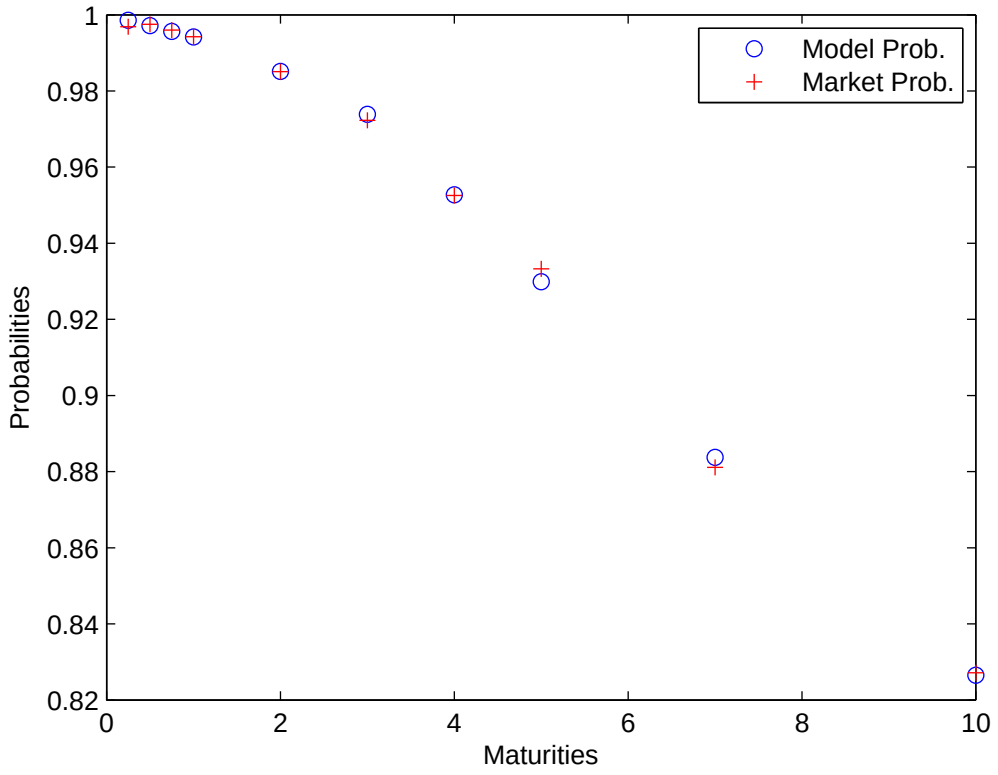

Figura 5.7: Survival probabilities con coefficienti costati a tratti, dopo la doppia calibrazione su 10 valori di mercato

T	Prob. modello	Prob. mercato	Err. relativo	Err. assoluto
0.25	0.998597	0.99689	0.17128%	0.17075%
0.5	0.997155	0.99752	0.03654%	0.03645%
0.75	0.995673	0.99598	0.03078%	0.03066%
1	0.994188	0.99425	0.00622%	0.00618%
2	0.985116	0.98508	0.00368%	0.00363%
3	0.973882	0.9723	0.16267%	0.15816%
4	0.952721	0.95254	0.01901%	0.01810%
5	0.929887	0.93328	0.36354%	0.33929%
7	0.88374	0.88112	0.29734%	0.26199%
10	0.826488	0.82716	0.08130%	0.06725%

Tabella 5.7: Survival probabilities con coefficienti costati a tratti, dopo la doppia calibrazione su 10 valori di mercato

Notiamo che con l'ultima calibrazione fatta su 40 dati di mercato, l'errore relativo medio si è abbassato rispetto alla prima calibrazione in cui i coefficienti sono costanti in tutto l'intervallo $[0, 4]$, da 0.188534% a 0.136203%; l'errore assoluto medio si è abbassato da 0.166227% a 0.120249%.
Per quanto riguarda la calibrazione fatta su 10 valori di mercato realmente quotati, l'errore relativo medio è sceso da 0.164079% a 0.117236%, quello assoluto medio è sceso da 0.149225% a 0.109246%.
Nel complesso, abbiamo ottenuto degli errori molto piccoli.

L'obiettivo principale della tesi era quello di presentare un modello stocastico per la valutazione delle opzioni che tenesse conto del rischio di credito, e di calibrarlo tramite il metodo Monte Carlo. Solo con il calcolo sequenziale, non saremmo stati in grado di raggiungere questo obiettivo.
Abbiamo quindi dimostrato la potenza del calcolo in parallelo attraverso CUDA, e la sua possibile applicazione in ambito finanziario.

Bibliografia

[1] Tomas Bjork. *Arbitrage theory in continuous time.* Oxford university press, 2009.

[2] Damiano Brigo. *Constant maturity credit default swap valuation with market models*, 2006.

[3] Damiano Brigo and Aurelien Alfonsi. *Credit default swap calibration and derivatives pricing with the ssrd stochastic intensity model. Finance and stochastics*, 2005.

[4] Damiano Brigo and Fabio Mercurio. *Interest rate models-theory and practice: with smile, inflation and credit.* Springer Science Business Media, 2007.

[5] Damiano Brigo and Massimo Morini. *Cds market formulas and models. In Invited presentation at XVIII Warwick Option Conference*, 2005.

[6] Peter Carr and Vadim Linetsky. *A Jump to Default Extended CEV Model: An Application of Bessel Processes*, 2006.

[7] John Cheng, Max Grossman, Ty McKercher. *PROFESSIONAL CUDA® C Programming*, 2014.

[8] Fuchang Gao, Lixing Han. *Implementing the Nelder-Mead simplex algorithm with adaptive parameters*, 2010.

[9] Paul Glasserman. *Monte Carlo methods in financial engineering. Applications of mathematics.* Springer, 1 edition, 2004.

[10] John C Hull and Alan White. *Valuing credit default swaps i: No counterparty default risk*, 2000.

[11] Monique Jeanblanc and Marek Rutkowski. *Default risk and hazard process. In Mathematical FinanceBachelier Congress 2000.* Springer, 2002.

[12] Andrea Pascucci. *PDE and Martingale methods in option pricing.* Springer-Verlag Italia, Milano, 2011.

[13] Yu Tian, Zili Zhu, Fima Klebaner, Kais Hamza. *Calibrating and Pricing with Stochastic-Local Volatility Model*, 2007.