Politecnico di Milano
Scuola di Ingegneria industriale e dell'informazione

Corso di laurea magistrale in Ingegneria matematica

Progetto d'esame per il corso di
Programmazione Avanzata per il Calcolo Scientifico

Estrazione della Densità degli Stati in semiconduttori organici

Studente:
Pasquale Claudio Africa
Matricola 816884

Docenti:
Luca Formaggia
Carlo De Falco

Anno Accademico 2013–2014

Indice

Elenco delle figure

Lista degli acronimi

MIS Metal-Insulator-Semiconductor

DOS Density of States

EGDM Extended gaussian disorder model

BIM Box Integration Method

HOMO Highest occupied molecular orbital

LUMO Lowest unoccupied molecular orbital

PCG Preconditioned conjugate gradient

BiCGSTAB Biconjugate gradient stabilized method

CSV Comma-separated values

P(NDI2OD-T2) Poly{[N,N'-bis(2-octyldodecyl)-1,4,5,8-naphthalene dicarboximide-2,6-diyl]-alt-5,5'-(2,2'-bithiophene)}

0 Introduzione

0.1 Abstract

Lo scopo del presente lavoro è lo sviluppo di una libreria in linguaggio C++ per la simulazione di modelli relativi a dispositivi elettronici.
Si considera una specifica classe di dispositivi basati su semiconduttori di natura organica (polimeri), costituita dai condensatori di tipo Metal-Insulator-Semiconductor (MIS), il cui nome mette in evidenza la loro struttura «a strati».
La piena comprensione dei fenomeni che avvengono al loro interno ha come requisito fondamentale la conoscenza della funzione Density of States (DOS), dal momento che questa quantità concorre a determinare le proprietà dei materiali optoelettronici e le loro prestazioni. Per ricavare questa funzione è stato risolto un **problema di identificazione dei parametri**: noto il modello che descrive il comportamento di questi dispositivi in dipendenza da alcuni parametri (non misurabili sperimentalmente) è possibile, effettuando delle ipotesi sulla forma della DOS, verificare che i risultati ottenuti siano coerenti con i dati sperimentali a disposizione.

Il progetto si colloca nell'ambito di una collaborazione tra il laboratorio *MOX*, il *Dipartimento di Elettronica, Informazione e Bioingegneria* del **Politecnico di Milano** e il *Center for Nano Sciences and Technology* dell'**Istituto Italiano di Tecnologia**.
Le simulazioni riguardano dunque dispositivi realistici, le cui caratteristiche sono ancora oggetto di studio. Ciò ha concesso infine la possibilità di effettuare un fitting dei valori simulati su dati sperimentali raccolti in laboratorio (relativi a misure accoppiate di tensione e capacità elettrica), garantendo una maggiore accuratezza dei risultati.

0.2 Motivazione

Il sistema drift-diffusion:

$$\begin{cases} -\nabla \cdot (\epsilon \nabla \varphi) = \rho & \text{in } \Omega \\ q\dfrac{\partial n}{\partial t} - q\nabla \cdot (D_n \nabla n - n\mu_n \nabla \varphi) + qR = qG & \text{in } \Omega_{semic} \\ q\dfrac{\partial p}{\partial t} - q\nabla \cdot (D_p \nabla p + p\mu_p \nabla \varphi) + qR = qG & \text{in } \Omega_{semic} \end{cases}$$

dove:

- ϵ è la permittività elettrica del mezzo $\left[C \cdot V^{-1} \cdot m^{-1}\right]$;
- q è il quanto di carica elettrica $[C]$;
- φ è potenziale elettrico $[V]$;
- ρ è la densità di carica $\left[C \cdot m^{-3}\right]$;
- n e p sono le concentrazioni volumetriche di elettroni e lacune $\left[m^{-3}\right]$;
- $D_{n,p}\left[m^{-2} \cdot s^{-1}\right]$ e $\mu_{n,p}\left[V^{-1} \cdot m^{-2} \cdot s^{-1}\right]$ sono i coefficienti di diffusività e di mobilità dei portatori rispettivamente;
- $(R - G)$ è il tasso netto di ricombinazione–generazione $\left[m^{-3} \cdot s^{-1}\right]$;
- Ω è l'intero dominio;
- Ω_{semic} è la porzione di dominio relativa al solo semiconduttore.

descrive l'andamento spaziale e temporale del potenziale elettrico e delle concentrazioni di elettroni e lacune in un generico dispositivo a semiconduttore, tenendo conto degli effetti diffusivi (generati da una concentrazione non uniforme di cariche nel dispositivo) e convettivi (dovuti alla presenza di un campo elettrico $-\nabla\varphi$ che, per la legge di Coulomb, genera una forza che trasporta le cariche).
Si supponga che si possa applicare un modello Extended gaussian disorder model (EGDM)[Coe+05; Pas+05] per le mobilità:

$$\mu_n(T, \nabla\varphi, n) = \mu_0(T) \cdot g_1(\nabla\varphi) \cdot g_2(n)$$

e che valga la relazione di Einstein-Smoluchowski generalizzata[Liu+11]:

$$\frac{D_n}{\mu_n} = g_3 V_{th}$$

(analogamente per le lacune), dove g_1, g_2 sono funzioni e g_3 una costante dipendenti dal materiale e V_{th} è la tensione di soglia (verrà definita a pagina 5).
In particolare, $g_1(\cdot)$ e $g_2(\cdot)$ sono dipendenti implicitamente da un parametro σ relativo al disordine molecolare (e quindi legato alla DOS): si tratta di un valore non misurabile sperimentalmente ma che, una volta noto, permette al modello drift-diffusion di descrivere *in toto* il comportamento del dispositivo dal punto di vista molecolare. La dipendenza dei coefficienti di diffusività e mobilità da σ fa emergere dunque la necessità di un'analisi che porti a stimare σ (seppur in maniera non esatta) risolvendo un opportuno problema di identificazione dei parametri: ciò permetterebbe di avere un modello matematico che descriva con maggiore precisione i fenomeni fisici coinvolti. Un'estensione naturale di questo progetto consiste nel risolvere un problema di **ottimizzazione multi-obiettivo**, considerando la dipendenza da altri parametri non strettamente legati a σ (ad esempio derivanti dalla combinazione di più DOS di forme diverse).

0.3 Descrizione

Il codice sviluppato è un'estensione di un già esistente pacchetto di algoritmi scritti in linguaggio `Octave` per la risoluzione del problema e intende migliorarne le prestazioni.
È stata considerata una sezione monodimensionale del dispositivo, per garantire un buon trade-off tra costi computazionali e aderenza alla realtà; essa è costituita da:

- un contatto metallico (back), mantenuto al potenziale di terra;
- uno strato di semiconduttore, di spessore t_{semic};
- uno strato di isolante, di spessore t_{ins};
- un contatto metallico (gate), a cui viene imposta una tensione dall'esterno.

La coordinata z di riferimento assume valori nell'intervallo $[-t_{semic}, t_{ins}]$. Il modello matematico, costituito da un'equazione di Poisson integro-differenziale non

lineare, è stato prima linearizzato attraverso un metodo di Newton e successivamente discretizzato attraverso la variante Box Integration Method (BIM) nella famiglia dei metodi ai volumi finiti. Segue infine una fase di post-processing, in cui i risultati della simulazione vengono fittati sui dati sperimentali del dispositivo in esame.
Sono stati implementati due casi test:

- `simulate_dos`, che simula il modello, fitta i risultati sui dati sperimentali e salva i dati e i plot relativi per poter valutare la bontà della simulazione;
- `fit_dos`, che invoca il simulatore all'interno di un ciclo di ottimizzazione; durante l'esecuzione viene individuato il valore ottimale (cioè per il quale la distanza in norma H^1 dai dati sperimentali sia minima) del parametro di disordine σ all'interno di un range specificato.

0.4 Sommario

Nel primo capitolo verrà discussa la fisica del problema; in particolare verranno ricavate le equazioni e discusse le relazioni costitutive che descrivono il modello matematico.

Nel secondo capitolo si illustreranno brevemente i metodi utilizzati per la risoluzione del modello, dalla linearizzazione fino alla formulazione algebrica.

Nel terzo capitolo argomento centrale saranno aspetti più pratici relativi all'implementazione in C++, illustrando le tecniche di programmazione e i vari tool utilizzati.

Nel quarto capitolo verranno illustrati i risultati di casi test (ottenuti a partire da diverse condizioni dei parametri di input) e presentate infine alcune possibilità di estensione del presente lavoro.

1 Modello

1.1 Equazione di Poisson

Le note equazioni di Maxwell:

$$\nabla \cdot \vec{D} = \rho \tag{1.1a}$$

$$\nabla \times \vec{E} + \frac{\partial \vec{B}}{\partial t} = \vec{0} \tag{1.1b}$$

$$\nabla \cdot \vec{B} = 0 \tag{1.1c}$$

$$\nabla \times \vec{H} - \frac{\partial \vec{D}}{\partial t} = \mathbf{J} \tag{1.1d}$$

dove:

- $\vec{E}$ è il vettore campo elettrico $\left[V \cdot m^{-1}\right]$;
- $\vec{D}$ è il vettore induzione elettrica $\left[C \cdot m^{-2}\right]$ (se la permittività ϵ non varia nel tempo vale la relazione costitutiva: $\vec{D} = \epsilon_0 \epsilon_r \vec{E} = \epsilon \vec{E}$, dove ϵ_0 è la permittività nel vuoto $\left[C \cdot V^{-1} \cdot m^{-1}\right]$ e ϵ_r, adimensionale, quella relativa del mezzo);
- ρ è la densità di carica elettrica $\left[C \cdot m^{-3}\right]$;
- $\vec{B}$ è il vettore induzione magnetica $\left[N \cdot A \cdot m^{-1}\right]$;
- $\vec{H}$ è il vettore induzione magnetica nei materiali $\left[A \cdot m^{-1}\right]$;
- $\mathbf{J}$ è la densità superficiale di corrente elettrica $\left[A \cdot m^{-2}\right]$;

descrivono l'evoluzione spazio-temporale del campo elettromagnetico in un mezzo in presenza di una densità di carica o di una corrente. Dalla (1.1c), è possibile introdurre un *potenziale vettore*, cioè un campo vettoriale $\vec{A}$ tale che:

$$\nabla \times \vec{A} = \vec{B}\,.$$

Si noti che un vettore nella forma $\vec{A} + \nabla\phi$ (ottenuto cioè tramite il **Gauge di Lorenz**[Jac99]), con ϕ arbitraria è ancora un potenziale vettore per $\vec{B}$, essendo il rotore di un gradiente sempre pari al vettore nullo; si sfrutterà in seguito questa informazione per scegliere un potenziale vettore che abbia una divergenza opportuna. Sotto

alcune ipotesi di regolarità sulle funzioni e supponendo il dominio semplicemente connesso, vale, sostituendo la precedente relazione nella (1.1b):

$$\nabla \times \left(\vec{E} + \frac{\partial \vec{A}}{\partial t} \right) = \vec{0} \Rightarrow \vec{E} + \frac{\partial \vec{A}}{\partial t} = -\nabla \varphi$$

per una funzione φ opportuna definita *potenziale scalare*.
Ricordando il legame tra campo elettrico e campo di induzione elettrica:

$$\vec{D} + \epsilon \frac{\partial \vec{A}}{\partial t} = -\epsilon \nabla \varphi$$

Applicando l'operatore divergenza ad entrambi i membri dell'ultima relazione e scambiando le derivate in tempo con quelle in spazio:

$$\nabla \cdot \vec{D} + \epsilon \frac{\partial}{\partial t} (\nabla \cdot \vec{A}) = -\nabla \cdot (\epsilon \nabla \varphi)$$

da cui, sfruttando la (1.1a):

$$\epsilon \frac{\partial}{\partial t} (\nabla \cdot \vec{A}) + \nabla \cdot (\epsilon \nabla \varphi) = -\rho \, .$$

Essendo ϕ arbitraria, è possibile sceglierla in modo da soddisfare la **condizione di Lorenz**:

$$\epsilon \frac{\partial}{\partial t} (\nabla \cdot \vec{A}) = -\frac{1}{c^2} \frac{\partial \varphi}{\partial t}$$

dove c è la velocità di propagazione delle onde elettromagnetiche nel vuoto. Si ottiene dunque l'equazione delle onde per φ:

$$\frac{1}{c^2} \frac{\partial^2 \varphi}{\partial t^2} - \nabla \cdot (\epsilon \nabla \varphi) = \rho \, .$$

Essendo generalmente c molto maggiore delle velocità caratteristiche del mezzo considerato (in questo caso, ad esempio, delle velocità medie con cui si muovono i portatori di carica), il primo termine è trascurabile rispetto agli altri. Si ottiene in definitiva l'**equazione di Poisson** per il potenziale elettrostatico:

$$\boxed{-\nabla \cdot (\epsilon \nabla \varphi) = \rho} \, . \tag{1.2}$$

Le derivate che compaiono nella precedente equazione sono da intendersi in senso debole, essendo ϵ una funzione non continua; assume infatti valori costanti a

tratti, in particolare:

$$\epsilon = \begin{cases} \epsilon_{semic} & \text{nel semiconduttore} \\ \epsilon_{ins} & \text{nell'isolante.} \end{cases}$$

1.2 Semiconduttori organici

Dal punto di vista microscopico, il comportamento dei semiconduttori è ben descritto dal *modello a bande*: gli atomi che formano un materiale sono costituiti da più «strati» (o bande o livelli[1]) energetici, occupati da elettroni; a strati più esterni corrispondono livelli di energia più alti. Nei **conduttori** generalmente l'ultima banda non è totalmente riempita, e questo permette agli elettroni di muoversi facilmente dando origine ad una corrente.
In un materiale organico, l'ultimo strato pienamente occupato viene definito **Highest occupied molecular orbital (HOMO)** (equivalente alla banda di valenza nei materiali inorganici), mentre il primo strato non pienamente occupato prende il nome di **Lowest unoccupied molecular orbital (LUMO)** (l'equivalente della banda di conduzione).

Quello che accade in un **semiconduttore** è che la struttura degli orbitali HOMO e LUMO è strettamente legata alla temperatura e, maggiormente, al grado di disordine molecolare: se un elettrone è dotato di energia sufficiente, è permesso che «salti» dallo strato HOMO allo strato LUMO, il quale per definizione non sarà del tutto occupato; in questa condizione gli elettroni saranno liberi di muoversi e generare una corrente. Più alta sarà la temperatura o il grado di disordine, più il moto degli elettroni sarà favorito. Un elettrone che si sposta dalla sua posizione lascia quella che nel linguaggio della fisica viene chiamata ***lacuna*** (o buca; in inglese: *hole*), termine che indica l'assenza di un elettrone; la meccanica quantistica spiega che le lacune sono delle *quasi-particelle*, a cui viene associata la stessa carica dell'elettrone (in valore assoluto); hanno quindi carica positiva ma una massa diversa da quella dell'elettrone e che dipende dal materiale considerato; un'altra proprietà è che le

[1] Nella modellizzazione matematica un «livello» è un vettore di uno spazio di Hilbert complesso che è rappresentato, in una base opportuna dello spazio (ad esempio quella degli stati con posizione ben definita), da una *funzione d'onda*; la funzione d'onda racchiude tutte le informazioni circa lo stato quantistico di una particella che si trova su quel livello.

lacune hanno una mobilità minore di quella elettronica. Elettroni e lacune vengono chiamati *portatori di carica*. Intuitivamente, una corrente di elettroni in una direzione dà origine ad una corrente di lacune in verso opposto; per questo motivo, nell'analisi dei fenomeni legati a questo comportamento, bisogna tener conto dei due effetti congiunti, sommandoli.

In condizioni di equilibrio termodinamico[GP05] (cioè quando ogni processo è bilanciato dal suo inverso) la concentrazione $p\ [m^{-3}]$ di lacune uguaglia la concentrazione n di elettroni: questo valore viene definito come **concentrazione di intrinseci** n_i. In formule: $np = n_i n_i = n_i^2$, che prende il nome di **legge dell'azione di massa**. Per aumentare la conducibilità del semiconduttore si può agire ad esempio attraverso il cosiddetto **drogaggio**, tecnica utilizzata in particolare nel caso dei semiconduttori inorganici. Con questo termine si intende l'aggiunta all'intrinseco di un certo numero di atomi di natura diversa (anche detti *impurità*), generalmente con un elettrone in più (*donori*) o in meno (*accettori*) rispetto al semiconduttore: si parla rispettivamente di drogaggio di **tipo n** con concentrazione N_D o di **tipo p** con concentrazione N_A; le concentrazioni di drogaggio vengono solitamente assunte costanti nel tempo. La struttura elettronica di un materiale drogato consente a elettroni e lacune maggiore libertà di movimento. Oltre certe soglie di drogaggio il semiconduttore può *degenerare*, nel senso che può assumere comportamenti del tutto simili a quelli dei materiali conduttori.
La densità di carica che rientra nella (1.2) assume dunque la forma:

$$\rho = \begin{cases} -q\,(n - p + N_A - N_D) & \text{nel semiconduttore} \\ 0 & \text{nell'isolante.} \end{cases} \tag{1.3}$$

1.3 Statistica di Fermi-Dirac

È possibile stimare il numero di elettroni e lacune presenti negli strati HOMO e LUMO attraverso dei modelli che derivano dalla meccanica statistica[Sac].

Definizione 1. Per **livello di Fermi** $\mathcal{E}_F$ si intende il livello occupato di maggior energia in un sistema di fermioni alla temperatura di 0K.

Con abuso di linguaggio, si indicherà con l'espressione «livello di Fermi» sia il livello di maggior energia sia l'energia ad esso associata.

Definizione 2. La **statistica di Fermi-Dirac** è la distribuzione statistica identificata dalla funzione densità:

$$f_D(\mathcal{E}) = \frac{1}{1+\exp\left(\frac{\mathcal{E}-\mathcal{E}_F}{k_B \cdot T}\right)}$$

dove $k_B = 1.38 \cdot 10^{-23}\left[J \cdot K^{-1}\right]$ è la costante di Boltzmann e T la temperatura del materiale. Tale densità, tenendo conto del principio di esclusione di Pauli[2], descrive la probabilità che un elettrone abbia un'energia pari a $\mathcal{E}$ (e indica che quel livello è occupato); la probabilità invece che una lacuna abbia energia pari a $\mathcal{E}$ è pari a $(1 - f_D(\mathcal{E}))$.

I **fermioni** sono le particelle che seguono la distribuzione di Fermi-Dirac (fra le quali protoni, neutroni ed elettroni).

Osservazione 1. Per come è stato definito, il livello di Fermi è tale per cui

$$f_D(\mathcal{E}_F) = \frac{1}{2}\,.$$

Se $\mathcal{E} - \mathcal{E}_F \gg k_B \cdot T$ allora si può effettuare l'approssimazione:

$$f_D(\mathcal{E}) \approx \exp\left(-\frac{\mathcal{E}-\mathcal{E}_F}{k_B \cdot T}\right)$$

che corrisponde invece alla **statistica di Maxwell-Boltzmann**, spesso utilizzata nei materiali inorganici.

Osservazione 2. Il fattore $V_{th} := \frac{k_B \cdot T}{q}$, dove q è il quanto di carica elettrica, è definito **tensione di soglia** (vale circa $26mV$ alla temperatura ambiente di 300K). Esso ha un ruolo importante nella fisica dei semiconduttori: semplificando a titolo di maggiore chiarezza, si può dire che generalmente i dispositivi rimangono «spenti» finché la tensione applicata ai loro capi non supera la tensione di soglia.

Siano $\mathcal{E}_{LUMO}$ e $\mathcal{E}_{HOMO}$ i livelli energetici dei rispettivi orbitali molecolari (funzioni del potenziale elettrico applicato φ) e sia $\mathcal{E}_{gap} := \mathcal{E}_{LUMO} - \mathcal{E}_{HOMO}$ l'*energy gap* tra i due livelli.
Sia $g(\mathcal{E})$ la funzione che descrive la DOS, cioè la densità degli stati quantici possibili aventi energia $\mathcal{E}$. L'intervallo $[\mathcal{E}_{HOMO}, \mathcal{E}_{LUMO}]$ determina la **banda proibita**, dove cioè $g(\cdot)$ è nulla. Dalla leggi della meccanica quantistica, è possibile ricavare le

[2] in base al quale lo stesso livello può essere occupato al più da un fermione.

seguenti relazioni per le concentrazioni:

$$n = \int_{-\infty}^{+\infty} g(\mathcal{E} - \mathcal{E}_{LUMO}) \cdot f_D(\mathcal{E} - \mathcal{E}_F)\, d\mathcal{E} \tag{1.4a}$$

$$p = \int_{-\infty}^{+\infty} g(\mathcal{E}_{HOMO} - \mathcal{E}) \cdot [1 - f_D(\mathcal{E} - \mathcal{E}_F)]\, d\mathcal{E} \tag{1.4b}$$

dove l'integrale è esteso a tutti i livelli di energia ammissibili. Queste espressioni hanno un'immediata interpretazione: corrispondono a delle medie integrali al variare di tutti i livelli di energia possibili, dove la probabilità che un elettrone (rispettivamente una lacuna) abbia una certa energia $\mathcal{E}$ è pesata sul numero di stati possibili aventi quella stessa energia in riferimento a $\mathcal{E}_{LUMO}$ (rispettivamente $\mathcal{E}_{HOMO}$). Nel seguito della trattazione verranno assunte le seguenti ipotesi:

- il semiconduttore è intrinseco (cioè il drogaggio è nullo, come spesso avviene nel caso organico); in questo caso la (1.3) nel semiconduttore diventa:

$$\rho = -q(n - p)$$

- gli effetti termici sono trascurabili, cioè $\mathcal{E}_{gap}$ può essere considerato sufficientemente grande;

- le correnti di dispersione nello strato isolante sono nulle;

- il regime di evoluzione è quasi-statico: il dispositivo è sempre in equilibrio termodinamico e fenomeni di trasporto non vengono coinvolti; il livello di Fermi $\mathcal{E}_F$ è dunque costante in tutto il dispositivo e può essere considerato pari a $0\,J$, senza ledere la generalità;

- solo una specie di portatori di carica è presente nel dispositivo: per semplicità, si suppone che la concentrazione p di lacune sia nulla e che gli effetti dipendano solo dalla presenza degli elettroni: il caso opposto può essere trattato in maniera del tutto analoga; allora la (1.3) può essere ulteriormente semplificata, nella zona del semiconduttore, in:

$$\boxed{\rho = -qn}\,. \tag{1.5}$$

  Questa ipotesi corrisponde ad una realtà molto diffusa nel caso dell'elettronica organica: dispositivi in cui entrambi i portatori diano contributi rilevanti al

trasporto di carica sono molto difficili da ottenere; presentano inoltre un livello di disordine molecolare così elevato da mettere in discussione la validità del modello.

Infine, essendo il potenziale elettrico φ definito a meno di una costante additiva, è possibile scegliere un livello di riferimento che permetta di scrivere:

$$\varphi = -\frac{\mathcal{E}_{\text{LUMO}}}{q} \tag{1.6}$$

nella zona del semiconduttore.

1.4 Relazioni costitutive per la Density of States

Un tema molto delicato e che non trova un consenso univocamente riconosciuto in letteratura è quale sia la relazione costitutiva più adatta a descrivere la funzione DOS $g(\cdot)$ che compare nelle (1.4) nei materiali «disordinati» (come quelli a base organica). In questo lavoro si è supposto che $g(\cdot)$ appartenga a una famiglia di funzioni parametrizzate da un singolo parametro (indicato con σ o λ) che rappresenta il **grado di disordine** del sistema. Diversi modelli sono stati proposti, tra cui:

1. una gaussiana simmetrica[FT09; Poe+13; Mar+09];
2. una combinazione di gaussiane[Kwo+12];
3. un'esponenziale[VW11; Riv+11; RE11];
4. una gaussiana asimmetrica[TM11];
5. una combinazione di una gaussiana e un'esponenziale[Vri+13; Cho+14];
6. altro[VW09; Hul+04].

Il codice sviluppato consente di gestire i primi tre casi; il numero massimo di gaussiane che possono essere combinate è 4.

1.4.1 GAUSSIANA

La DOS di una singola gaussiana simmetrica assume la seguente forma:

$$g_\sigma(\cdot) = \frac{N_0}{\sqrt{2\pi}\sigma} e^{-\frac{(\cdot)^2}{2\sigma^2}}$$

dove:

- il parametro σ di disordine molecolare corrisponde alla deviazione standard;
- N_0 $\left(m^{-3}\right)$ è il numero totale di stati disponibili per unità di volume.

Sotto le assunzioni precedenti la (1.4a) si può scrivere come:

$$n = \frac{N_0}{\sqrt{2\pi}\sigma} \int_{-\infty}^{+\infty} \exp\left(-\frac{(\mathcal{E}-\mathcal{E}_{LUMO})^2}{2\sigma^2}\right) \frac{1}{1+\exp\left(\frac{\mathcal{E}}{k_B \cdot T}\right)} \, d\mathcal{E} \, . \tag{1.7}$$

L'obiettivo è scrivere la precedente relazione in una forma più facilmente calcolabile in fase di simulazione, ad esempio attraverso una formula di quadratura gaussiana. Attraverso la sostituzione:

$$\alpha = \frac{\mathcal{E}-\mathcal{E}_{LUMO}}{\sqrt{2}\sigma} \tag{1.8}$$

e grazie alla (1.6) si arriva a:

$$\boxed{n(\varphi) = \frac{N_0}{\sqrt{\pi}} \int_{-\infty}^{+\infty} e^{-\alpha^2} \left(1+\exp\left(\frac{\sqrt{2}\sigma\alpha - q\varphi}{k_B \cdot T}\right)\right)^{-1} d\alpha} \tag{1.9}$$

essendo $\mathcal{E} = \sqrt{2}\sigma\alpha + \mathcal{E}_{LUMO}$ e $d\mathcal{E} = \sqrt{2}\sigma d\alpha$.
È utile a questo punto calcolare la derivata di questa espressione rispetto al potenziale φ, che entrerà in gioco nell'applicazione del metodo di Newton presentato nel prossimo capitolo. Per far ciò, si rielabora la (1.7) utilizzando le espressioni riportate in precedenza:

$$n(\varphi) = \frac{N_0}{\sqrt{2\pi}\sigma} \int_{-\infty}^{+\infty} \exp\left(-\frac{(\mathcal{E}+q\varphi)^2}{2\sigma^2}\right) \frac{1}{1+\exp\left(\frac{\mathcal{E}}{k_B \cdot T}\right)} \, d\mathcal{E}$$

da cui:

$$\frac{dn}{d\varphi}(\varphi) = \frac{N_0}{\sqrt{2\pi}\sigma} \int_{-\infty}^{+\infty} \exp\left(-\frac{(\mathcal{E}+q\varphi)^2}{2\sigma^2}\right) \frac{1}{1+\exp\left(\frac{\mathcal{E}}{k_B \cdot T}\right)} \cdot \frac{-\cancel{2}(\mathcal{E}+q\varphi)}{\cancel{2}\sigma^2} \cdot q \; d\mathcal{E}$$

che diventa, ripetendo la sostituzione (1.8) e ricordando la (1.6):

$$\boxed{\frac{dn}{d\varphi}(\varphi) = -\frac{N_0 q}{\sigma}\sqrt{\frac{2}{\pi}}\int_{-\infty}^{+\infty} \alpha e^{-\alpha^2}\left(1+\exp\left(\frac{\sqrt{2}\sigma\alpha - q\varphi}{k_B\cdot T}\right)\right)^{-1} d\alpha}. \tag{1.10}$$

1.4.2 COMBINAZIONE DI GAUSSIANE

Nel combinare più gaussiane occorre specificare, per ciascuna di esse:

- $N_{0,i}$: il numero totale di stati disponibili per unità di volume;
- σ_i: il parametro di disordine (deviazione standard);
- $\varphi_{s,i}$: lo **shift** del potenziale elettrico rispetto alla prima gaussiana (per definizione, $\varphi_{s,1} = 0$);

dove i è un indice che identifica la gaussiana considerata.
L'ultimo di questi dati è una funzione **costante** nel dominio che si sommerà alla variabile φ presente nelle relazioni precedenti.
Se k è il numero di gaussiane considerate, la (1.9) si generalizza nel modo seguente:

$$\boxed{n(\varphi) = \sum_{i=1}^{k}\left(\frac{N_{0,i}}{\sqrt{\pi}}\int_{-\infty}^{+\infty} e^{-\alpha^2}\left(1+\exp\left(\frac{\sqrt{2}\sigma_i\alpha - q(\varphi+\varphi_{s,i})}{k_B\cdot T}\right)\right)^{-1} d\alpha\right)}$$

e, per la linearità dell'operatore derivata, la (1.10) diventa:

$$\boxed{\frac{dn}{d\varphi}(\varphi) = -\sum_{i=1}^{k}\left(\frac{N_{0,i}q}{\sigma_i}\sqrt{\frac{2}{\pi}}\int_{-\infty}^{+\infty} \alpha e^{-\alpha^2}\left(1+\exp\left(\frac{\sqrt{2}\sigma_i\alpha - q(\varphi+\varphi_{s,i})}{k_B\cdot T}\right)\right)^{-1} d\alpha\right)}.$$

1.4.3 ESPONENZIALE

Nel caso di DOS esponenziale vale la seguente relazione costitutiva[OHB12]:

$$g_\lambda(\cdot) = \frac{N_0}{\lambda}\exp\left(-\frac{(\cdot)}{\lambda}\right).$$

Anche in questo caso il parametro di disordine λ corrisponde alla deviazione standard.

La (1.4a) allora si può scrivere come:

$$n = \frac{N_0}{\lambda} \int_{\mathcal{E}_{LUMO}}^{+\infty} \exp\left(-\frac{(\mathcal{E} - \mathcal{E}_{LUMO})}{\lambda}\right) \frac{1}{1 + \exp\left(\frac{\mathcal{E}}{k_B \cdot T}\right)} \, d\mathcal{E} \, .$$

Attraverso il cambio di variabili:

$$\alpha = \frac{\mathcal{E} - \mathcal{E}_{LUMO}}{\lambda}$$

e grazie alla (1.6) si arriva a:

$$\boxed{n(\varphi) = \frac{N_0}{\lambda} \int_0^{+\infty} e^{-\alpha} \left(1 + \exp\left(\frac{\lambda\alpha - q\varphi}{k_B \cdot T}\right)\right)^{-1} d\alpha} \, . \tag{1.11}$$

Infine, con passaggi analoghi al caso della gaussiana, è facile mostrare che:

$$\boxed{\frac{dn}{d\varphi}(\varphi) = -\frac{N_0 q}{\lambda} \int_0^{+\infty} \alpha e^{-\alpha} \left(1 + \exp\left(\frac{\lambda\alpha - q\varphi}{k_B \cdot T}\right)\right)^{-1} d\alpha} \, . \tag{1.12}$$

1.5 Condizioni al contorno

Per risolvere l'equazione di Poisson descritta nelle sezioni precedenti, è necessario specificare delle opportune condizioni al contorno[Sel84] che garantiscano la buona posizione del problema. Per «contorno», in questo tipo di problemi, si intende generalmente l'insieme dei bordi relativi ai contatti elettrici; un dispositivo di tipo MIS, come quello preso in considerazione, presenta generalmente due *terminali* (che servono a collegarlo al resto del circuito), che sono:

- il **back**, indicato con Γ_{semic}, a contatto con il semiconduttore: viene mantenuto al potenziale di terra, cioè costituisce il riferimento rispetto al quale sarà calcolato il potenziale φ (rispettando opportunamente la (1.6));
- il **gate**, a contatto con l'isolante: su di esso viene imposta una tensione dall'esterno; la parte di frontiera da esso costituita verrà indicata con Γ_{ins}.

Essendo il potenziale, per quanto appena detto, fissato su entrambi, le condizioni imposte su di essi saranno di tipo Dirichlet. Prima di una descrizione dettagliata verranno introdotti alcuni concetti di interesse fisico.

Definizione 3. In elettronica dello stato solido, per **work-function** (W_f [J]) si intende la minima energia termodinamica necessaria per sottrarre un elettrone a un solido per portarlo nel vuoto immediatamente fuori dalla superficie del solido stesso, partendo dal livello di Fermi. Nel caso dei semiconduttori, la work-function dipende dall'eventuale drogaggio.

Definizione 4. L'**affinità elettronica** (indicata con E_a [J]) in fisica dei semiconduttori (da non confondere con l'affinità elettronica in chimica, con la quale coincide solo alla temperatura di 0K) è la minima energia termodinamica richiesta per far muovere un elettrone dall'estremo inferiore della banda di conduzione (livello LUMO) verso l'esterno del solido, nel vuoto; a differenza della work-function non dipende dal drogaggio.

1.5.1 BACK

Il fenomeno fisico considerato è la presenza di una **barriera di Schottky**[SM99], cioè una barriera energetica rilevata da elettroni e lacune all'interfaccia tra il semiconduttore e il metallo che costituisce il contatto elettrico. In particolare, se la natura del semiconduttore è organica, questo processo gioca un ruolo rilevante soprattutto nei dispositivi ottici (sensori luminosi oppure diodi/transistori a emissione di luce). Il metallo *inietta* (si parla di *iniezione termoionica*) elettroni e lacune nel semiconduttore; si dice che i portatori nel semiconduttore risentano dell'effetto coulombiano di una *carica immagine* derivante dai portatori nel metallo.
La condizione sul potenziale φ che modella questo tipo di fenomeno è[Sze81]:

$$\varphi = \phi_F - \phi_B \quad \text{su } \Gamma_{semic}$$

dove:

- ϕ_F è il potenziale di Fermi, cioè il livello di Fermi $\mathcal{E}_F$ convertito in potenziale (cioè diviso per $-q$); si ricorda che questo valore è stato posto convenzionalmente pari a 0J;
- ϕ_B è il potenziale di barriera, pari alla differenza tra la W_f del metallo di cui è costituito il contatto e la E_a del semiconduttore convertite in potenziali; è possibile dimostrare che questo valore è anche pari al potenziale corrispondende all'energia $\mathcal{E}_{LUMO}$ (ciò garantisce la coerenza con la (1.6)).

1.5.2 GATE

Sul contatto isolante viene imposta la condizione di Dirichlet:

$$\varphi = V_g + V_{shift} \quad \text{su } \Gamma_{ins}$$

dove V_g è il potenziale di gate (cioè il potenziale esterno applicato) e V_{shift} è un parametro che dipende da fattori come la presenza di dipoli permanenti, di cariche residue nell'isolante etc.

1.6 RIEPILOGO

Nel seguito della trattazione verrà considerato esclusivamente il caso in cui valga la relazione costitutiva gaussiana per la DOS (descritta in 1.4.1).
Fino a questo punto non è mai stata formulata alcuna ipotesi sulla geometria del sistema. Si considera adesso l'approssimazione monodimensionale precedentemente discussa sul dominio $\Omega = [-t_{semic}, t_{ins}]$.
Siano $\Omega_{semic} = (-t_{semic}, 0]$ e $\Omega_{ins} = (0, t_{ins})$ le porzioni di dominio relative al semiconduttore e all'isolante rispettivamente. Si ottiene dunque, a partire dalle (1.2), (1.5), (1.9), la seguente **equazione integro-differenziale** in φ:

$$\begin{cases} -\dfrac{d}{dz}\left(\epsilon \dfrac{d\varphi}{dz}\right)(z) = -\dfrac{N_0 q}{\sqrt{\pi}} \displaystyle\int_{-\infty}^{+\infty} e^{-\alpha^2} \left(1 + \exp\left(\dfrac{\sqrt{2}\sigma\alpha - q\varphi(z)}{k_B \cdot T}\right)\right)^{-1} d\alpha & z \in \Omega_{semic} \\ -\dfrac{d}{dz}\left(\epsilon \dfrac{d\varphi}{dz}\right)(z) = 0 & z \in \Omega_{ins} \\ \varphi(-t_{semic}) = -\phi_B \\ \varphi(t_{ins}) = V_g + V_{shift}\,. \end{cases} \tag{1.13}$$

Si sottolinea in particolare la forte **non-linearità** del problema (a causa del termine esponenziale), che richiederà l'applicazione di un metodo di Newton generalizzato. La seconda equazione garantisce che, nello strato isolante, il potenziale φ varia linearmente e questo permette di ottenere una buona guess iniziale per il metodo numerico di risoluzione, ad esempio una funzione lineare in tutto il dispositivo che agli estremi del dominio assuma il valore delle condizioni al bordo.

2 Metodi numerici

Nel presente capitolo verranno illustrati i metodi numerici utilizzati per trattare le equazioni presentate nella 1.6.

2.1 Linearizzazione: il metodo di Newton

L'equazione di Poisson (1.13) è, come già detto, non lineare a causa della presenza del termine esponenziale (in particolare nella zona del semiconduttore).
È possibile riscriverla in questi termini, ricordando la (1.7):

$$-\frac{d}{dz}\left(\epsilon\frac{d\varphi}{dz}\right)(z)+\frac{N_0 q}{\sqrt{2\pi}\sigma}\int_{-\infty}^{+\infty}\exp\left(-\frac{(\mathcal{E}+q\varphi(z))^2}{2\sigma^2}\right)\frac{1}{1+\exp\left(\frac{\mathcal{E}}{k_B\cdot T}\right)}\,d\mathcal{E}=0\,,$$

espressione che mette in evidenza una struttura del tipo $\mathcal{F}(\varphi)=0$, dove $\mathcal{F}$ è un funzionale integro-differenziale (cioè un operatore da un opportuno spazio di Hilbert che garantisca il soddisfacimento delle condizioni al bordo in $\mathbb{R}$); questa forma suggerisce l'applicazione di un metodo di Newton generalizzato, che prevede la risoluzione di:

$$\begin{cases}\mathcal{DF}\left(\varphi^{(k)}\right)[\delta\varphi^{(k)}]=-\mathcal{F}\left(\varphi^{(k)}\right) & \text{(2.1a)}\\ \varphi^{(k+1)}=\varphi^{(k)}+\delta\varphi^{(k)} & \text{(2.1b)}\end{cases}$$

ad ogni iterazione k fino a convergenza.

Osservazione 1. Si noti che, essendo $\mathcal{F}$ un *operatore* e non una funzione, il simbolo $\mathcal{DF}(\varphi)[\chi]$ denota la *derivata funzionale* di F, calcolata nel «punto» φ e valutata in corrispondenza della funzione χ.

Questa derivata può essere calcolata in senso *debole* secondo la definizione di **Gâteaux** nel modo seguente:

$$\mathcal{DF}(\varphi)[\chi]=\lim_{\kappa\to 0}\frac{\mathcal{F}(\varphi+\kappa\chi)-\mathcal{F}(\varphi)}{\kappa}=$$

(sostituendo l'espressione di $\mathcal{F}$):

$$= \lim_{\kappa\to 0} \frac{1}{\kappa}\left[-\frac{d}{dz}\left(\epsilon\frac{d\varphi}{dz} + \kappa\epsilon\frac{d\chi}{dz}\right) + \frac{d}{dz}\left(\epsilon\frac{d\varphi}{dz}\right)\right] +$$
$$+ \lim_{\kappa\to 0} \frac{1}{\kappa}\frac{N_0 q}{\sqrt{2\pi}\sigma}\int_{-\infty}^{+\infty}\left[\exp\left(-\frac{(\mathcal{E}+q(\varphi+\kappa\chi))^2}{2\sigma^2}\right) - \exp\left(-\frac{(\mathcal{E}+q\varphi)^2}{2\sigma^2}\right)\right]\cdot$$
$$\cdot\frac{1}{1+\exp\left(\frac{\mathcal{E}}{k_B\cdot T}\right)}\,d\mathcal{E} =$$

(effettuando uno sviluppo di Taylor per $\kappa \sim 0$):

$$= -\frac{d}{dz}\left(\epsilon\frac{d\chi}{dz}\right) + \frac{N_0 q}{\sqrt{2\pi}\sigma}\int_{-\infty}^{+\infty}\exp\left(-\frac{(\mathcal{E}+q\varphi)^2}{2\sigma^2}\right)\cdot$$
$$\cdot \underbrace{\lim_{\kappa\to 0}\frac{1}{\kappa}\left[\exp\left(-\frac{q^2\kappa^2\chi^2 + 2(\mathcal{E}+q\varphi)\kappa\chi}{2\sigma^2}\right) - 1\right]}_{\simeq -\frac{\cancel{q^2\kappa\chi^2}^{\,0} + 2(\mathcal{E}+q\varphi)\chi}{2\sigma^2}}\frac{1}{1+\exp\left(\frac{\mathcal{E}}{k_B\cdot T}\right)}\,d\mathcal{E} =$$
$$= -\frac{d}{dz}\left(\epsilon\frac{d\chi}{dz}\right) - \frac{N_0 q^2\chi}{\sqrt{2\pi}\sigma}\int_{-\infty}^{+\infty}\exp\left(-\frac{(\mathcal{E}+q\varphi)^2}{2\sigma^2}\right)\frac{\mathcal{E}+q\varphi}{2\sigma^2}\frac{1}{1+\exp\left(\frac{\mathcal{E}}{k_B\cdot T}\right)}\,d\mathcal{E} =$$

(sostituendo la (1.5), la (1.8) e la (1.10) ed esplicitando le dipendenze funzionali):

$$= -\frac{d}{dz}\left(\epsilon\frac{d\chi}{dz}\right)(z) + q\frac{dn}{d\varphi}(\varphi)\chi =$$
$$= -\frac{d}{dz}\left(\epsilon\frac{d\chi}{dz}\right)(z) - \frac{d\rho}{d\varphi}(\varphi)\chi\,.$$

La (2.1a) diventa quindi:

$$-\frac{d}{dz}\left(\epsilon\frac{d\left(\delta\varphi^{(k)}\right)}{dz}\right) - \frac{d\rho}{d\varphi}\left(\varphi^{(k)}\right)\cdot\delta\varphi^{(k)} = -\left(-\frac{d}{dz}\left(\epsilon\frac{d\left(\varphi^{(k)}\right)}{dz}\right) - \rho\left(\varphi^{(k)}\right)\right) \tag{2.2}$$

che è un'equazione differenziale in $\delta\varphi^{(k)}$, in cui compare un termine di diffusione (che darà origine, nella discretizzazione, a una matrice di stiffness) e un termine di reazione (che darà origine a una matrice di massa). In questa espressione, ρ è valutata a partire dalla (1.9).

Per garantire che $\varphi^{(k)}$ assuma ad ogni iterazione k le condizioni al bordo imposte dalla (1.13), è necessario accoppiare quest'equazione con le seguenti condizioni di

Dirichlet:

$$\begin{cases} \delta\varphi^{(k)}(-t_{semic}) = 0 \\ \delta\varphi^{(k)}(t_{ins}) = 0\,. \end{cases}$$

Una volta discretizzata la (2.1a) con un metodo generico (per esempio agli elementi o ai volumi finiti etc.) e fissata una guess iniziale $\vec{\varphi}_0$ (per la quale una possibile scelta è quella descritta nella sezione 1.6), occorrerà risolvere la sequenza di sistemi lineari del metodo di Newton:

$$\begin{cases} \left(\mathcal{K} - \mathcal{M}\cdot \mathrm{diag}\left(d\vec{\rho}^{(k)}\right)\right)\delta\vec{\varphi}^{(k)} = \mathcal{K}\cdot\vec{\varphi}^{(k)} - \mathcal{M}\cdot\mathrm{diag}\left(\vec{\rho}^{(k)}\right) \\ \vec{\varphi}^{(k+1)} = \vec{\varphi}^{(k)} + \delta\vec{\varphi}^{(k)} \end{cases} \tag{2.3}$$

ad ogni iterazione k fino a convergenza, dove:

- $\delta\vec{\varphi}^{(k)}$ è una opportuna discretizzazione di $\delta\varphi^{(k)}$ (ad esempio in corrispondenza dei nodi della griglia);
- $\mathcal{K}$ è la matrice di stiffness del metodo di discretizzazione considerato, che include il contributo del termine di permittività ϵ;
- $\mathcal{M}$ è la matrice di massa sottoposta a **lumping**[Qua08], ottenuta approssimando gli integrali con la regola di quadratura dei trapezi;
- $\mathrm{diag}(\vec{v})$ è la matrice che ha gli elementi del vettore $\vec{v}$ sulla diagonale e 0 altrove;
- $\vec{\rho}^{(k)}$ è una discretizzazione di $\rho\left(\varphi^{(k)}\right)$;
- $d\vec{\rho}^{(k)}$ è una discretizzazione di $\dfrac{d\rho}{d\varphi}\left(\varphi^{(k)}\right)$.

La convergenza del metodo di Newton è verificata attraverso la norma infinito della differenza tra due iterate successive; il numero massimo di iterazioni e la soglia di tolleranza possono essere impostati dall'utente.

2.2 Discretizzazione: il metodo BIM

La discretizzazione dell'equazione linearizzata (2.1) è stata effettuata attraverso il metodo BIM[BCC98]. La famiglia dei metodi ai volumi finiti (a cui il BIM appartiene) è spesso (ma non esclusivamente) utilizzata per discretizzare equazioni scritte

in forma conservativa, cioè espressioni del tipo: $\frac{\partial u}{\partial t} + \nabla \cdot \vec{F}(u) = s(u)$, dove u è la variabile (scalare o meno) che «si conserva», $\vec{F}$ e s sono delle funzioni assegnate ed indicano rispettivamente il flusso, che può essere un campo vettoriale o tensoriale, e la sorgente del campo); queste equazioni, se integrate su un *volume di controllo*, danno origine ad una formulazione integrale della legge di conservazione del flusso.

2.2.1 APPLICAZIONE ALL'EQUAZIONE DI POISSON

Nel caso dell'equazione linearizzata (2.2) in esame, la variabile «conservata» è $\delta\varphi^{(k)}(z)$; siano per semplicità $f^{(k)} = \frac{d\rho}{d\varphi}\left(\varphi^{(k)}\right)$ e $s^{(k)}$ il right-hand side dell'equazione, entrambe funzioni della coordinata z (dipendenza che verrà sottintesa per non aggravare ulteriormente la notazione).

L'applicazione del metodo BIM, che richiede almeno che la mesh utilizzata sia conforme (l'intersezione tra due elementi della griglia può essere al più un vertice o un intero lato, mai una porzione di esso), consiste in questi step[Sha99]:

1. creazione dei box;
2. scrittura del problema in forma integrale locale su ogni singolo box;
3. ipotesi che il flusso sia costante su ogni lato dei box e assemblaggio finale.

1) COSTRUZIONE DEI BOX I **box** costituiscono una nuova geometria del dominio ottenuta dalla mesh iniziale; comunemente, considerando ad esempio il caso di una mesh triangolare bidimensionale, si ricava con uno dei seguenti metodi:

- per ogni triangolo si unisce il circocentro (punto di intersezione degli assi) al punto medio di ogni lato (*box di Voronoi*);
- per ogni triangolo si unisce il baricentro (punto di intersezione delle mediane) al punto medio di ogni lato (*box baricentrici*).

I box sono pertanto le superfici delimitate dalle spezzate chiuse così ottenute.
Nella geometria monodimensionale considerata, il dominio di riferimento viene partizionato in n sotto-domini di lunghezza uguale $h = \frac{1}{n}$; questi costituiscono i «volumi di controllo» $\mathcal{B}_i$, centrati nel punto medio z_i di ciascun intervallo. Siano inoltre $z_{i-\frac{1}{2}}$ e $z_{i+\frac{1}{2}}$ le coordinate degli estremi (rispettivamente sinistro e destro) dell'i-esimo box.

2) FORMULAZIONE INTEGRALE LOCALE Il problema differenziale iniziale può essere ridotto a un sistema di n equazioni in forma integrale locale, ciascuna su un singolo box. In un metodo ai volumi finiti le incognite approssimano la media integrale della soluzione in ogni cella, secondo la formula:

$$(u)_i \approx \frac{1}{h}\int_{z_{i-\frac{1}{2}}}^{z_{i+\frac{1}{2}}} u(z)\, dz \tag{2.4}$$

dove $(u)_i$ è assunta costante nel box i-esimo (può essere considerata come un'incognita nodale fissata in z_i), in contrasto con le differenze finite, che approssimano invece i valori puntuali della soluzione, e con gli elementi finiti, dove si approssimano i coefficienti delle combinazioni lineari delle funzioni di base.
Integrando dunque l'equazione:

$$-\frac{d}{dz}\left(\epsilon\frac{d\left(\delta\varphi^{(k)}\right)}{dz}\right) - f^{(k)}\delta\varphi^{(k)} = s^{(k)}$$

sul box i-esimo si ottiene:

$$-\int_{z_{i-\frac{1}{2}}}^{z_{i+\frac{1}{2}}}\frac{d}{dz}\left(\epsilon\frac{d\left(\delta\varphi^{(k)}\right)}{dz}\right)\, dz - \int_{z_{i-\frac{1}{2}}}^{z_{i+\frac{1}{2}}} f^{(k)}\delta\varphi^{(k)} dz = \int_{z_{i-\frac{1}{2}}}^{z_{i+\frac{1}{2}}} s^{(k)}\, dz\,.$$

Questa procedura può essere interpretata come un metodo agli elementi finiti in cui sia scelta come funzione test l'indicatrice di ciascun intervallo (o elemento, in più dimensioni).
Indicando per semplicità con $F_{i\pm\frac{1}{2}} = -\epsilon(z_{i\pm\frac{1}{2}})\frac{d\left(\delta\varphi^{(k)}\right)}{dz}(z_{i\pm\frac{1}{2}})$ l'approssimazione numerica del flusso attraverso i bordi $z_{i-\frac{1}{2}}$ e $z_{i+\frac{1}{2}}$ del box e dividendo la precedente equazione per il passo di discretizzazione h si ottiene, usando l'approssimazione introdotta nella (2.4):

$$\frac{F_{i+\frac{1}{2}} - F_{i-\frac{1}{2}}}{h} - \left(f^{(k)}\delta\varphi^{(k)}\right)_i = \left(s^{(k)}\right)_i\,.$$

3) IPOTESI DI FLUSSO COSTANTE E ASSEMBLAGGIO FINALE L'incognita $\delta\varphi^{(k)}$ compare ancora sotto il segno di derivata; occorre quindi esplicitare una relazione che dipenda dai valori nodali $\{\varphi_i\}_{i=1}^{n}$. Per far ciò, è possibile ad esempio

utilizzare degli schemi di questo tipo:

$$\frac{d\left(\delta\varphi^{(k)}\right)}{dz}(z_{i-\frac{1}{2}}) \approx \frac{\delta\varphi_i^{(k)} - \delta\varphi_{i-1}^{(k)}}{h}$$
$$\frac{d\left(\delta\varphi^{(k)}\right)}{dz}(z_{i+\frac{1}{2}}) \approx \frac{\delta\varphi_{i+1}^{(k)} - \delta\varphi_i^{(k)}}{h} .$$

Sostituendo queste relazioni nella precedente equazione e attraverso dei semplici passaggi algebrici si giunge a:

$$-\frac{\epsilon(z_{i+\frac{1}{2}})\left(\delta\varphi_{i+1}^{(k)} - \delta\varphi_i^{(k)}\right) - \epsilon(z_{i-\frac{1}{2}})\left(\delta\varphi_i^{(k)} - \delta\varphi_{i-1}^{(k)}\right)}{h^2} - \left(f^{(k)}\delta\varphi^{(k)}\right)_i = \left(s^{(k)}\right)_i$$

per ogni $i = 2,\ldots,n-1$. Si tratta di un sistema lineare di $n-2$ equazioni nelle $n-2$ incognite nodali $\delta\varphi_i^{(k)}$; si ricorda che al bordo sono state assunte condizioni di Dirichlet, che hanno permesso di eliminare 2 gradi di libertà dal sistema.
Tuttavia, la scelta seguita è stata quella presentata da D. Scharfetter e H. Gummel in [SG69], relativa a un più generale problema di diffusione-trasporto e molto robusta per la simulazione di problemi relativi a dispositivi elettronici a semiconduttore[FRB83]; è possibile, attraverso degli opportuni cambi di variabile[Sac], giungere ad una formula alle differenze che lega le derivate di $\delta\varphi^{(k)}$ con i suoi valori nodali. Si perviene dunque alla formulazione algebrica presentata a pagina 15.

2.3 Post-processing

Dopo la risoluzione dell'equazione di Poisson nel potenziale φ, vengono calcolate la carica elettrica totale presente e la capacità C [F] del dispositivo (che per piccoli segnali può essere modellata come la capacità equivalente di due condensatori in serie, rispettivamente il semiconduttore e l'isolante).
Segue una fase di post-processing, in cui i risultati vengono confrontati con i dati sperimentali a disposizione: la curva $C(V_g)$ (dove V_g è un range discreto di tensioni applicate entro il quale si vuole simulare il dispositivo, che copre dal regime di funzionamento in *depletion*, o svuotamento, $V_{g,dep}$ a quello in accumulazione[GP05] $V_{g,acc}$) viene shiftata di un potenziale V_{shift}, pari alla distanza (sulle ascisse) dei picchi della curva $\frac{dC}{dV_g}(V_g)$ valutata a partire dai dati sperimentali (C_{exp}) e dai

risultati della simulazione (C_{sim}).
Vengono calcolati gli errori in norma L^2 e H^1 tra le due curve e la distanza tra i massimi (sulle ordinate) della loro derivata e, infine, il **centro di carica** [m], funzione del potenziale di gate V_g, definito come:

$$t_{CoC}(V_g) = \frac{\int_{-t_{semic}}^{0} \rho(z) z \, dz}{\int_{-t_{semic}}^{0} \rho(z) \, dz}$$

il cui valore può essere utilizzato per calcolare la capacità equivalente del semiconduttore ($C_{semic} = A\frac{\epsilon_{semic}}{t_{CoC}}$, dove A è l'area della sezione del dispositivo).

2.4 Fitting automatico

I problemi legati alla non-idealità degli effetti considerati introduce nel modello delle incertezze non direttamente quantificabili, che influenzano sensibilmente il valore del parametro d'interesse σ. Il primo problema riguarda il potenziale V_{shift} (che compare nelle condizioni al bordo): da un punto di vista sperimentale esso tiene conto di molti fenomeni che possono causare uno shift della curva $C(V_g)$ rispetto al potenziale, ad esempio la presenza di dipoli permanenti o di cariche residue nell'isolante, una non corretta modellazione della work-function del gate etc. Non è possibile prevedere l'entità dell'influenza che questi fattori esercitano nel modello, né è possibile effettuare delle misurazioni sperimentali che la quantifichino: tutti questi fenomeni pertanto vengono racchiusi implicitamente nel potenziale di shift.

Un secondo effetto da tenere presente è che la capacità elettrica totale del dispositivo non è soltanto la serie di quella del semiconduttore e dell'isolante, ma è soggetta a contributi di capacità parassita dovuti, per esempio, a un «accoppiamento» tra il gate e il back oppure a linee di campo elettrico che uniscono i due contatti etc. Questi contributi vengono modellati con un condensatore di capacità C_{sb} (*stray capacitance*) idealmente connesso in parallelo al MIS.

Infine, la misura dello spessore t_{semic} del semiconduttore è soggetta a incertezze a causa delle tecniche sperimentali impiegate.
È stato dunque implementato, nel file test `fit_dos.cc`, un algoritmo iterativo di fitting che permette di identificare il vettore di parametri $\tilde{X} = [\sigma, C_{sb}, t_{semic}]$

ottimale. La tensione V_{shift}, noto un generico $\tilde{X}$, può essere calcolata come segue:

$$V_{shift} = \arg\max_{V_g} \frac{dC_{sim}}{dV_g}(V_g;\tilde{X}) - \arg\max_{V_g} \frac{dC_{exp}}{dV_g}(V_g)$$

Le fasi di cui si compone ogni passo del ciclo di ottimizzazione (che viene eseguito per un numero di iterazioni fissato dall'utente) sono:

STEP 1 Si inizializza un range discreto di valori centrati in $\sigma^{(k)}$, per ciascuno dei quali verrà eseguita una simulazione. Fissati $C_{sb}^{(k)}$ e $t_{semic}^{(k)}$ si individua dunque la $\sigma^{(k+1)}$ identificando la simulazione che avrà generato l'errore (in una qualche norma specificata dall'utente) minore rispetto ai dati sperimentali.

STEP 2 Si aggiorna C_{sb}:

$$C_{sb}^{(k+1)} = C_{sb}^{(k)} + C_{exp}(V_{g,acc}) - C_{sim}\left(V_{g,acc};\left[C_{sb}^{(k)}, t_{semic}^{(k)}, \sigma^{(k+1)}\right]\right)$$

STEP 3 Si aggiorna t_{semic}:

$$t_{semic}^{(k+1)} = \epsilon_{semic}\left(\frac{1}{C_{exp}(V_{g,dep}) - C_{sb}^{(k+1)}} - \frac{t_{ins}}{\epsilon_{ins}}\right)$$

le ultime due espressioni derivano da formule relative alla capacità di un condensatore MIS in regime di svuotamento e di accumulazione[GP05]). Questo algoritmo è sostenuto soltanto dall'evidenza e dalla consistenza dei risultati ottenuti, mentre dal punto di vista teorico non sono stati effettuati studi approfonditi che dimostrino le sue proprietà di convergenza oppure, ad esempio, quale sia la scelta più prudente sul criterio d'arresto da adottare.

2.5 Formule di quadratura

Gli integrali che compaiono nelle (1.9), (1.10), (1.11), (1.12) possono essere approssimati numericamente attraverso delle formule di quadratura. Sono state utilizzate in particolare due regole gaussiane, in cui cioè sia i nodi sia i pesi di quadratura sono incogniti.

2.5.1 GAUSS-HERMITE

Permette di approssimare l'integrale monodimensionale di una funzione $f(z)$ pesata su un termine $g(z)$ di tipo gaussiano:

$$\int_{-\infty}^{+\infty} e^{-z^2} f(z)\, dz\,.$$

I nodi in una formula di grado n sono definiti come gli zeri del n-esimo polinomio di Hermite H_n, definito dalla relazione ricorsiva:

$$H_{n+1}(z) = 2zH_n(z) - 2nH_{n-1}(z)$$

dove $H_0(z) = 1, H_{-1}(z) = 0$.

2.5.2 GAUSS-LAGUERRE

In questo caso si considera un peso $g(z)$ di tipo esponenziale negativo:

$$\int_{0}^{+\infty} e^{-z} f(z)\, dz\,.$$

Analogamente, i nodi corrispondono agli zeri dei polinomi di Laguerre:

$$L_{n+1}(z) = \frac{2n+1-z}{n+1} L_n(z) - \frac{n}{n+1} L_{n-1}(z)$$

dove $L_0(z) = 1, L_{-1}(z) = 0$.
Sia N_g il numero di nodi desiderato per l'approssimazione. Per entrambi i metodi occorre trovare i nodi $\{z_i\}_{i=1}^{N_g}$ e i pesi $\{w_i\}_{i=1}^{N_g}$ tali che l'integrale si approssimi come:

$$\int g(z) f(z)\, dz \approx \sum_{i=1}^{N_g} w_i \cdot f(z_i)\,.$$

Esistono procedure di diversa natura che permettono di calcolare i nodi e i pesi di quadratura; in particolare per ciascuna delle due regole, nella libreria in esame, sono stati implementati:

- un algoritmo basato sulla decomposizione agli autovalori/autovettori della matrice di Jacobi[KU98] $J \in \mathbb{R}^{N_g \times N_g}$ (tridiagonale), metodo valido per qualsiasi famiglia di polinomi definiti da una relazione ricorsiva che goda di certe proprietà sui coefficienti; i nodi di quadratura corrispondono dunque agli

autovalori di J e i pesi agli autovettori normalizzati su un termine che dipende dalla famiglia di polinomi considerata.

- un algoritmo di tipo iterativo[Pre+07], in cui si sfruttano un metodo di Newton e la relazione ricorsiva per calcolare le quantità d'interesse; la soglia di tolleranza e il numero massimo di iterazioni è fissato dall'utente. A differenza del precedente, garantisce prestazioni molto più efficienti, essendo coinvolte solo semplici operazioni algebriche e mai la fattorizzazione di matrici di grandi dimensioni.

2.6 Algoritmi e sistemi lineari

2.6.1 ALGORITMI NUMERICI

Nella stesura del codice, si è reso necessario fare ricorso ad algoritmi numerici per risolvere problemi di vario tipo; in particolare sono stati implementati algoritmi:

- di **sort**, per ordinare gli elementi presenti in un vettore; una versione modificata consente invece di tenere traccia delle posizioni precedenti all'ordinamento;
- di **interpolazione lineare**, utilizzato per interpolare un vettore y definito su una griglia x in corrispondenza di uno scalare o di un vettore x_{New} che non appartiene alla griglia iniziale;
- di **differenziazione numerica**, per calcolare la derivata di un vettore y rispetto a un altro vettore x delle stesse dimensioni: viene utilizzata una formula mista, che calcola una differenza in avanti nel primo nodo, una differenza all'indietro nell'ultimo e una differenza centrata nei restanti;
- di **integrazione numerica** attraverso il metodo di Newton-Cotes dei trapezi (utilizzata per scopi diversi da quelli presentati nella scorsa sezione); il calcolo degli errori in norma L^2 e H^1 è stato effettuato con questa tecnica.

2.6.2 RISOLUZIONE DEI SISTEMI LINEARI

Una questione sempre aperta in analisi numerica è quale sia il metodo più adatto alla risoluzione di un sistema lineare; in particolare si fa riferimento al sistema (2.3).

È possibile dimostrare che, discretizzando l'equazione con il metodo BIM precedentemente descritto, tutte le matrici coinvolte sono **simmetriche e definite positive** (cioè il loro spettro è contenuto nel semiasse positivo della retta reale).

Per questo motivo, è stato possibile applicare un metodo basato sulla fattorizzazione di Cholesky generalizzata, cioè sulla decomposizione di una matrice nel prodotto LDL^T, dove L è una matrice triangolare inferiore e D una matrice diagonale, calcolabili attraverso un algoritmo efficiente che termina in un numero finito di passi. Attraverso sostituzioni successive, è possibile calcolare la soluzione del sistema lineare generico $LDL^T\vec{w} = \vec{b}$ splittandolo in tre fasi, nell'ordine:

$$\begin{aligned} &\text{i)} \quad L\vec{x} = \vec{b} \\ &\text{ii)} \quad D\vec{y} = \vec{x} \\ &\text{iii)} \quad L^T\vec{w} = \vec{y} \end{aligned}$$

che corrispondono a tre sistemi lineari risolubili esattamente in maniera performante viste le caratteristiche delle matrici coinvolte (un sistema triangolare inferiore può essere risolto in un numero di step pari alla sua dimensione). Tuttavia l'aritmetica *floating-point* dei calcolatori introduce degli errori di troncamento o di cancellazione numerica; per questo motivo la soluzione finale non sarà esatta ma approssimata. Un'altra scelta molto diffusa nel caso delle matrici s.d.p. è il metodo Preconditioned conjugate gradient (PCG); un buon precondizionatore può essere ottenuto a partire dalla fattorizzazione LU incompleta.

3 Implementazione di DosExtraction

In questo capitolo verrà illustrata la struttura del codice implementato, insieme ai vari tool di programmazione utilizzati in fase di *coding*.

3.1 Struttura del programma

La figura 3.1 illustra le relazioni tra i vari file utilizzati nel programma (comprese alcune librerie standard del linguaggio C++). Per utilizzare il codice implementato, è sufficiente includere l'header `src/dosModel.h`, che si curerà di includere automaticamente tutti gli altri file necessari. Il fulcro della libreria è costituito dal modulo `csvParser.h`, un parser di file di tipo Comma-separated values (CSV) che importa i dati sperimentali e i parametri di simulazione nelle varie strutture dati utilizzate dal programma. Il codice implementato è **exception-safe**: non appena si verifica un'eccezione, questa viene rilanciata a catena verso il `main`; da qui, dopo aver mostrato un messaggio d'errore informativo, il programma viene terminato cautamente attraverso l'istruzione `return EXIT_FAILURE`, lasciando il sistema operativo in uno stato sicuro. Le eccezioni vengono utilizzate per gestire casi anomali ma possibili (ad esempio la mancanza dei permessi in scrittura nella cartella di output o la non-convergenza di un algoritmo); per la gestione di casi impossibili invece (ad esempio vettori di dimensione errata o la chiamata a funzioni con parametri non appartenenti al range corretto) il controllo è effettuato attraverso l'istruzione `assert(condizione)` che informa l'utente sul punto esatto, nel codice, in cui si è verificato il problema (molto utile quindi in fase di debug). Definendo la macro `NDEBUG` questi controlli vengono ignorati. È fornito un sistema di configurazione automatica che utilizza il tool `CMake` (che sarà descritto nella prossima sezione); viene generata una libreria dinamica `libdosextraction.so` che dovrà essere linkata all'eseguibile: questo consente di non dover ricompilare tutti i file sorgente della libreria nel momento in cui si decida di sviluppare una nuova applicazione che sfrutti le funzionalità implementate. Nel `Makefile` generato è presente un target per installare l'eseguibile, la libreria e gli header file nel proprio sistema; l'utente può personalizzare le cartelle di destinazione in base al proprio utilizzo.

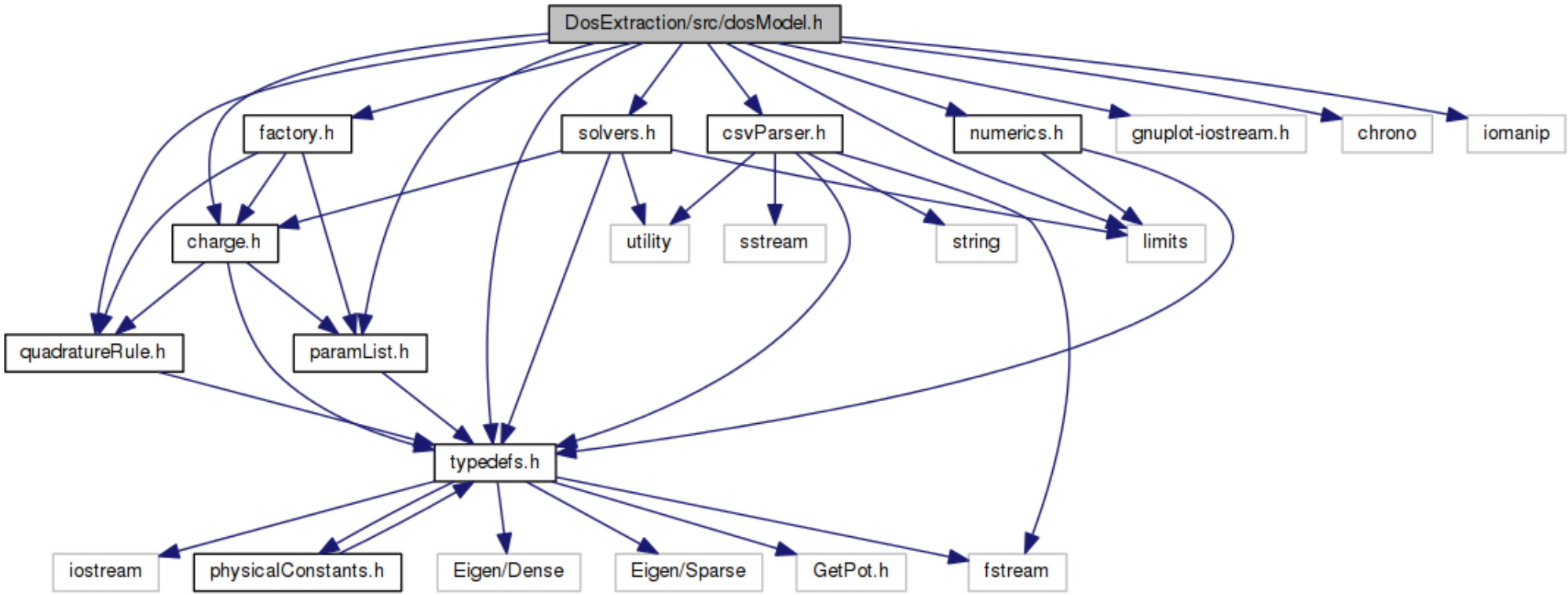

Figura 3.1: Struttura principale del programma

3.2 Librerie e tool di terze parti

Saranno adesso brevemente descritte le librerie di terze parti e i vari tool cui si è fatto ricorso nella stesura del codice.

3.2.1 EIGEN

Le `Eigen`[1] sono una libreria di algebra lineare molto potente. È composta esclusivamente da header file, trattandosi di una *template library*; i pro di una scelta di questo tipo sono: la non-necessità di dover compilare o installare la libreria (processo che, se il numero di file di cui si compone è molto grande, potrebbe richiedere molto tempo), una maggiore velocità di esecuzione (per la presenza di funzioni dichiarate `inline`) e una migliore portabilità del codice (che non dev'essere infatti riadattato per sistemi operativi diversi); i contro riguardano invece i seguenti punti: una modifica in una libreria header file richiede la ricompilazione di tutti i software che la utilizzano (al contrario ad esempio di una libreria linkata dinamicamente), i tempi di compilazione sono più lunghi (l'unità di compilazione deve controllare la definizione di tutti i componenti anziché solamente le loro interfacce) e la generazione di *code-bloat* (derivante dall'utilizzo di costrutti di tipo `inline`). All'interno di questa libreria si fa inoltre largo uso di tecniche di *template metaprogramming*.
Le `Eigen` definiscono delle classi per gestire strutture dati come vettori, matrici ad allocazione dinamica, sia in formato denso che sparso; è implementata inoltre una varietà di solutori per sistemi lineari, attraverso metodi diretti o iterativi, alcuni dei quali sono ottimizzati per lavorare su matrici sparse.
Una prima versione del codice era basata sulle librerie `Armadillo`[2], scelta in seguito abbandonata a causa della mancanza di solutori specifici per matrici sparse.
Infine, le `Eigen` sono state preferite ad altre librerie di algebra come `SuiteSparse`, `BLAS` o `LAPACK` per diverse ragioni: le prestazioni in un'esecuzione seriale (che è la condizione in cui sono state utilizzate nella libreria) sono in generale paragonabili o migliori, forniscono un'interfaccia notevolmente più immediata e supportata inoltre da una documentazione molto estesa e ricca di esempi; infine, le `Eigen` solo multipiattaforma tra i vari sistemi operativi, compilatori o hardware (non dovendo essere compilate o installate) e sono soggette a varie ottimizzazioni in fase di compilazione, processo che preserva comunque una durata ragionevole.

[1] http://eigen.tuxfamily.org/
[2] http://arma.sourceforge.net/

3.2.2 GETPOT

GetPot[3] è una libreria composta da un solo header file che implementa la classe omonima, la quale permette un semplice parsing delle opzioni specificate al programma da riga di comando e di file di configurazione direttamente nella fase di **runtime** (caratteristica importante per una maggiore astrazione e per evitare la necessità di ricompilare i sorgenti ad ogni modifica). In particolare, è stata utilizzata nei due file test per assolvere a un duplice compito:

- importare alcune impostazioni da un file di configurazione, di cui si riporta un breve esempio (da cui sono stati rimossi per brevità i commenti esplicativi):

```
1  input_params = TEST_1_Single_Sigma.csv
2
3  [QuadratureRule]
4      nNodes = 101
5      maxIterationsNo = 1000
6      tolerance = 1.0e-14
7
8  [NonLinearPoisson]
9      maxIterationsNo = 10
10     tolerance = 1.0e-4
```

- effettuare un parsing di comandi specificati al programma in fase di esecuzione, in particolare delle opzioni che specificano la directory e il nome del file di configurazione, indicati rispettivamente con le opzioni:
  `-d custom_directory_name -f custom_config_filename`

3.2.3 OPENMP

Il programma fornisce funzionalità di calcolo parallelo. Si è fatto ricorso alla libreria OpenMP[4] (per sistemi a memoria condivisa), che non richiede message passing ma gestisce in automatico la suddivisione dei compiti tra i vari thread. Inoltre non deve essere effettuata nessuna drastica modifica al codice seriale già scritto, è sufficiente specificare le opportune direttive che comunichino al preprocessore le impostazioni da utilizzare per parallelizzare il costrutto che le segue. Il numero di thread da utilizzare è impostato dall'utente nel file di configurazione GetPot: a

[3] http://getpot.sourceforge.net/
[4] http://openmp.org/

ciascuno di essi viene assegnata una simulazione (corrispondente in particolare a una riga nel file CSV di input dei parametri); il tutto viene gestito tramite *dynamic scheduling*: non appena un thread termina il compito che gli è stato assegnato, inizia subito a svolgerne un altro a partire dalla coda di esecuzione.
Nell'utilizzo di `OpenMP`, è necessario prestare particolare attenzione alla natura delle variabili, se dichiarate cioè private a ciascun thread o condivise tra tutti quelli coinvolti. In particolare l'oggetto `GetPot` che legge il file di configurazione deve essere dichiarato privato, altrimenti la condivisione tra più thread contemporaneamente causa un errore di tipo «`Segmentation fault.`» (probabilmente generato dal distruttore di `GetPot`) prima del termine del programma. Tuttavia, dovendo dichiarare un oggetto `GetPot` come `private` (istruzione che in `OpenMP` richiede che l'oggetto sia *default-constructable*), occorre che lo stesso sia reinizializzato da ogni thread (ciascuno dei quali deve infatti gestire lo stesso file). Altresì complicato è il sistema di gestione delle eccezioni all'interno di un *for-loop* parallelo: un'eccezione catturata da un thread deve necessariamente essere gestita dallo stesso e non può essere rilanciata all'interno del `catch`; questo problema è stato risolto come segue: il blocco `catch` relativo al ciclo `for` setta un booleano `ompThrewException` (condiviso tra i thread e inizializzato a `false`) al valore `true` e salva il messaggio d'errore in una stringa (anch'essa condivisa). Al termine del ciclo si verifica se un'eccezione è stata catturata; in caso positivo, viene stampato il messaggio d'errore e terminato il programma.

3.2.4 CMAKE

`CMake`[5] è un tool di configurazione e di gestione delle dipendenze molto potente. Mette a disposizione dei moduli per la ricerca, all'interno del proprio sistema, dei vari pacchetti richiesti. Per poter essere utilizzato, il progetto deve contenere il file `CMakeLists.txt`, che nel caso particolare si compone di queste parti:

- definizione di alcune variabili globali (ad esempio le directory di installazione, i nomi dei sorgenti, i nomi degli eseguibili da generare etc.);
- gestione delle librerie esterne; vengono incluse le cartelle `include/` e `src/` del progetto e individuati i seguenti pacchetti: `Eigen`, `OpenMP` (aggiungendo `-fopenmp` tra i flag del compilatore), `Gnuplot`, `Boost` (utilizzate dall'interfaccia C++ per `Gnuplot`), `Doxygen` e `AStyle`;

[5] http://www.cmake.org/

- definizione dei target di compilazione; vengono generati: la libreria `libdosextraction.so` a partire dai sorgenti, l'eseguibile `simulate_dos` e l'eseguibile `fit_dos`, entrambi ottenuti linkando la libreria appena creata;
- definizione dei target di installazione del software nel sistema operativo e relativa disinstallazione.

Un'alternativa molto valida a CMake è costituita dalla suite GNU Autotools[6] [7] [8], che presenta tuttavia una curva di apprendimento più bassa; inoltre CMake garantisce una portabilità migliore (anche verso sistemi operativi non Unix-like) e una maggiore velocità nella configurazione del sistema (lo script `configure` di Autotools è scritto in linguaggio bash e risulta pertanto più lento); da ultimo, il debug in ambiente CMake è in generale più intuitivo (gli script generati da Autotools contengono più informazioni e sono di più difficile comprensione); per questi motivi si è scelto in definitiva di adottare CMake. Le istruzioni per configurare il proprio sistema e compilare i sorgenti sono contenute nel file `README.md` disponibile nel pacchetto della libreria.

3.2.5 GNUPLOT

Gnuplot[9] è una utility per la generazione di plot. I suoi punti di forza sono la versatilità, la facilità di utilizzo che permette di ottenere un'ottima resa grafica in breve tempo e infine la disponibilità dell'interfaccia `gnuplot-iostream`[10] che permette di invocarlo dall'interno di un codice scritto in linguaggio C++, attraverso una classe di stream molto intuitiva, caratteristica ritenuta indispensabile e che ha portato a preferire Gnuplot rispetto ad altri strumenti dalle funzionalità simili. Grazie a questa libreria è possibile eseguire comandi definendo un oggetto di classe Gnuplot e utilizzando il relativo stream operator << che riceve in ingresso `const char *` oppure valori numerici. All'interno di `gnuplot-iostream` sono utilizzate alcune funzionalità implementate nei componenti `iostreams`, `system` e `filesystem` delle librerie Boost[11]. È stato utilizzato il terminale `pngcairo` per la generazione di grafici in formato png in alta qualità, insieme ad un layout `multiplot` che permette

[6] http://www.gnu.org/software/autoconf/
[7] http://www.gnu.org/software/automake/
[8] http://www.gnu.org/software/libtool/
[9] http://www.gnuplot.info/
[10] http://www.stahlke.org/dan/gnuplot-iostream/
[11] http://www.boost.org/

di suddividere la finestra in sottografici. Una funzionalità molto comoda di `Gnuplot` è la possibilità di ricevere in input i dati di cui generare il plot a partire da un file CSV (che è il formato utilizzato in questo progetto per il salvataggio dei risultati), importando automaticamente intere colonne di valori.

3.2.6 PROFILER E DEBUGGER

Per ottimizzare le prestazioni del software, sono stati utilizzati `Valgrind`[12] (una suite di tool per il profiling e il memory check) e il debugger `gdb`[13].

Il profiling è stato effettuato attraverso `Callgrind`: per poter essere utilizzato, è necessario compilare i sorgenti con un flag che produca le informazioni di debug (ad esempio `-g` nel caso di `GCC` e `Clang`); durante il processo di profiling vengono salvate informazioni relative al peso con cui **ciascuna** funzione invocata contribuisce al tempo totale di esecuzione. Quest'analisi ha fatto emergere in particolare la seguente questione: la chiamata a metodi *getter* (o *accessor*) era molto onerosa vista la frequente necessità di restituire i parametri per copia, in particolare relativamente alle classi `Charge` (ad esempio nel metodo `n_approx`), `QuadratureRule` e `NonLinearPoisson-Solver1D` (e derivate); per ottenere uno *speedup* sono state inserite all'interno di esse delle relazioni di *friendship* con le classi che maggiormente utilizzano i loro attributi: ciò rende loro accessibili tutti i membri privati, evitando di dover ricorrere ai metodi getter. Si evidenzia anche che, dal punto di vista dell'interfaccia pubblica, è più conveniente che i metodi getter restituiscano i loro valori per *const reference* anziché per copia: un `const_cast` potrebbe rimuovere l'attributo `const` e quindi permettere una modifica non sicura, ma nel caso di oggetti complessi (quali ad esempio vettori e matrici ad allocazione dinamica della memoria) è preferibile incorrere in questo rischio piuttosto che avere un programma molto meno performante a causa delle costose chiamate alla funzione `malloc()`. È bene sottolineare, tuttavia, che, se l'intestazione di un metodo è: `const type & Class::get()`, il risparmio computazionale si ha soltanto invocandolo, se ciò è possibile, attraverso: `const type & var = Class::get()` anziché tramite: `type var = Class::get()`, essendo in quest'ultimo caso effettuata comunque una copia. I risultati del profiling sono stati analizzati attraverso lo strumento di visualizzazione `KCacheGrind`.

È stato inoltre invocato il tool `Memcheck` per verificare lo stato della memoria durante l'esecuzione del programma, ad esempio un consumo non richiesto a cau-

[12] http://valgrind.org/
[13] http://www.gnu.org/software/gdb/

sa di *memory leak* (memoria non liberata oppure file non chiusi dopo l'utilizzo). Quest'analisi non ha evidenziato particolari problematiche.

Lo strumento `gdb` (che esegue il programma senza rallentarlo per effettuare altre operazioni, come nel caso di `Valgrind`), invece, ha permesso l'identificazione della parte di codice generante l'errore «`Segmentation fault.`» nella regione parallelizzata (come descritto nella precedente 3.2.3); molto utile, in particolare, è stata la funzione `backtrace`: essa fornisce informazioni su come il programma è giunto nello stato in cui si è interrotto (stampando la catena delle invocazioni alle varie funzioni). In particolare l'invocazione ad alcune funzioni della libreria `OpenMP` ha suggerito la parte di codice in cui controllare la presenza di errore.

3.2.7 GIT

Per il presente progetto è stato utilizzato il sistema di controllo di versione `git`[14], che permette di tenere traccia delle modifiche apportate al codice durante il suo sviluppo in maniera veloce ed efficiente. Il progetto è disponibile in un repository privato hostato da `GitHub` e disponibile all'indirizzo: `https://github.com/elauksap/DosExtraction` (in cui sono riportate le istruzioni per compilare i sorgenti e generare la documentazione). In particolare è stato utilizzato il servizio fornito da `GitHub` di gestione delle release, che permette di associare una release del progetto ad un determinato tag (o ad un commit, nel caso di un *lightweight tag*); le release possono essere scaricate in un archivio `zip` o `tar.gz`.

3.2.8 DOXYGEN

I sorgenti del progetto sono stati documentati attraverso il *lexical scanner* `Doxygen`[15]; si tratta di uno strumento semplice quanto potente: inserendo all'interno del codice dei commenti scritti nella sintassi propria di `Doxygen`, è possibile generare la documentazione completa in formato `html` oppure `pdf` tramite LaTeX; nel primo caso, è stata scelta l'opzione che permette di consultare i sorgenti direttamente dall'interno della documentazione, permettendo all'utente una visione d'insieme sul codice e i relativi riferimenti ipertestuali alle varie classi, per una più facile comprensione. La cartella di output può essere impostata nel file `CMakeLists.txt` prima della compilazione: attraverso `CMake` verrà creato un file di configurazione

[14] `http://git-scm.com/`
[15] `http://www.stack.nl/~dimitri/doxygen/`

per Doxygen in base alle preferenze dell'utente. È inoltre definito un target per la generazione della documentazione, che può avvenire attraverso il comando:

```
$ make doc
```

digitato dopo aver configurato il proprio sistema con CMake.

3.2.9 ARTISTIC STYLE

Per mantenere una coerenza stilistica tra i vari file del progetto, si è fatto ricorso al piccolo tool AStyle[16] che, attraverso delle impostazioni scelte dallo sviluppatore, adatta i sorgenti specificati nel formato scelto. Le linee guida di riferimento sullo stile sono quelle di Google[17], utilizzate insieme allo stile di indentazione Allman. Per precauzione, i file originali vengono mantenuti ma rinominati con il suffisso .orig. È stato definito in CMake un target che esegue automaticamente questo tool prima della compilazione dei sorgenti; può essere invocato anche manualmente attraverso il comando:

```
$ make astyle
```

3.3 TYPEDEFS GLOBALI

I file physicalConstants.h e typedefs.h contengono la definizione di alcuni *typedefs* e costanti fisiche e numeriche globali (definite all'interno di un namespace), comprendenti ad esempio la carica dell'elettrone q, la costante di Boltzmann k_B, la permittività elettrica nel vuoto ϵ_0, il π etc.

```
1 namespace constants
2 {
3     const Real     Q = 1.60217653000000e-19;
4     const Real    Q2 = Q * Q                ;
5     const Real   K_B = 1.38065050000000e-23;
6     const Real T_REF = 300                  ;
7     const Real  KB_T = K_B * T_REF          ;
8     const Real  EPS0 = 8.854187817e-12      ;
9 }
```

```
1 typedef double Real;
```

[16] http://astyle.sourceforge.net/
[17] http://google-styleguide.googlecode.com/svn/trunk/cppguide.html

```
2  typedef ptrdiff_t Index;
3
4  using namespace Eigen;
5
6  typedef Matrix<Real, Dynamic, Dynamic> MatrixXr   ;
7  typedef Matrix<Real, Dynamic,       1> VectorXr   ;
8  typedef Matrix<Real,       1, Dynamic> RowVectorXr;
9  typedef SparseMatrix<Real>             SparseXr   ;
10
11 template<typename ScalarType>
12 using VectorX = Matrix<ScalarType, Dynamic, 1>;
13
14 template<typename T>
15 using VectorXpair = VectorX<std::pair<T, Index> >;
16
17 namespace constants
18 {
19     const Index PARAMS_NO = 27;
20
21     const Real      PI = M_PI              ;
22     const Real SQRT_PI = std::sqrt(PI)     ;
23     const Real   PI_M4 = 0.7511255444649425;
24     const Real  SQRT_2 = std::sqrt(2)      ;
25 }
```

Il tipo `ptr_diff` è quello utilizzato di default nella libreria `Eigen` per l'indicizzazione all'interno di vettori e matrici (sono consentiti valori negativi, essendo un tipo *signed*). I simboli del namespace `Eigen` vengono resi disponibili all'interno di tutto il codice, non essendo presenti ambiguità con altri nomi. Vengono definiti dei typedef abbreviativi per indicare matrici (sia nel formato sparso che denso) e vettori, aventi dimensione dinamica, di numeri reali; inoltre si è fatto ricorso ai **type alias** (o *alias template*) del `C++11` per definire dei tipi in dipendenza da un parametro template. La variabile `PARAMS_NO` definisce il numero **esatto** di parametri che devono essere specificati nei file dei parametri letto dalla classe `CsvParser`.

3.4 La classe CsvParser

La classe `CsvParser` è il cardine delle funzionalità della libreria. Vista la necessità di lettura dei parametri del modello a partire da file CSV numerici, si è reso necessario utilizzare un modulo che permettesse di farlo in maniera agevole ed efficiente; tuttavia, le (poche) librerie open-source disponibili in rete non garantivano un tale livello di astrazione o un'interfaccia con le librerie `Eigen` utilizzate, pertanto è stata interamente codificata una nuova struttura dati.
Vengono riportati di seguito i metodi e attributi **principali** della classe.

```
1  class CsvParser
2  {
3      public:
4          CsvParser() = delete;
5          CsvParser(const std::string &, const bool & = true);
6          virtual ~CsvParser();
7
8          RowVectorXr importRow(const Index &);
9          MatrixXr    importRows(const std::initializer_list<Index> &);
10         ...
11         VectorXr    importCol(const Index &);
12         MatrixXr    importCols(const std::initializer_list<Index> &);
13         ...
14         Real        importCell(const Index &, const Index &);
15         MatrixXr    importAll();
16
17     private:
18         ...
19         std::ifstream input_;
20         ...
21         char separator_;
22 };
```

Il costruttore di default è stato dichiarato *deleted* (funzionalità dello standard C++11: un'alternativa consiste nel dichiararlo `private`) dal momento che l'oggetto non può essere costruito senza indicare una stringa contenente il nome del file ed eventualmente un booleano opzionale per specificare se la prima riga del file dei parametri contiene le «etichette» dei valori e dev'essere pertanto ignorata; il costruttore apre dunque il file di input (`ifstream`) in modalità di lettura e individua il separatore testuale in un set che comprende la virgola, un carattere di tabulazione, i

due punti o un semplice spazio. I metodi principali vengono utilizzati per importare una o più righe o colonne dal file e salvarli all'interno di vettori riga o colonna nel formato denso delle `Eigen`; vengono utilizzati parametri di tipo lista di inizializzazione per l'import multiplo (invocato ad esempio con `importRows({1,3,4})`; anche questa è una funzionalità dello standard 11 del linguaggio). Infine sono offerti dei metodi per l'import di una singola cella o dell'intero file. L'indicizzazione avviene, escludendo eventualmente la prima riga, a partire dal numero 1.
Attraverso la funzione `assert()` viene verificato se si sta tentando di accedere ad un indice di riga o di colonna non appartenente ai bound (letti attraverso il costruttore) del file considerato.

3.5 La classe ParamList

Questa classe si occupa di gestire e mantenere in memoria una lista di parametri che definiscono il modello da simulare.

```
1  class ParamList
2  {
3      public:
4          friend class     GaussianCharge;
5          friend class ExponentialCharge;
6          friend class           DosModel;
7
8          ParamList() = default;
9          explicit ParamList(const RowVectorXr &);
10         virtual ~ParamList() = default;
11         ...
12 };
```

Le classi `DosModel` e quelle derivate da `Charge` (la relazione di *friendship* non viene ereditata) sono dichiarate `friend` per avere pieno accesso ai parametri privati (che vengono utilizzati per esempio anche all'interno di loop con un numero elevato di iterazioni), senza dover ricorrere a metodi getter che rallenterebbero di molto l'esecuzione del programma. I parametri (privati) vengono importati a partire da un vettore riga (ad esempio importato tramite `CsvParser`) e devono essere specificati **esattamente** nel seguente ordine:

1) l'indice della simulazione;

2), 3) lo spessore del semiconduttore; lo spessore dello strato isolante;

4), 5) la permittività elettrica relativa del semiconduttore; la permittività elettrica dell'isolante (verranno convertite in permittività assolute al momento dell'import);

6) la temperatura a cui si vuole effettuare la simulazione;

7), 8) la work-function del contatto di back e l'affinità elettronica del semiconduttore (convertite in energia al momento dell'import);

9), 10) N_0, σ, relativi a una DOS gaussiana (σ verrà moltiplicata per $k_B \cdot 300K$ al momento dell'import);

11), 12), 13) $N_{0,2}$, σ_2, $\varphi_{s,2}$, relativi alla seconda gaussiana;

14), 15), 16) $N_{0,3}$, σ_3, $\varphi_{s,3}$, relativi alla terza gaussiana;

17), 18), 19) $N_{0,4}$, σ_4, $\varphi_{s,4}$, relativi alla quarta gaussiana;

20), 21) $N_{0,exp}$, λ_{exp}, relativi a una DOS esponenziale (λ verrà moltiplicata per $k_B \cdot 300K$ al momento dell'import);

22), 23) l'area del semiconduttore e la guess per la capacità parassita C_{sb};

24) il numero di nodi in cui suddividere la mesh;

25), 26), 27) il numero di step in cui suddividere il range di tensioni, l'estremo inferiore e superiore del range.

La classe fornisce dei metodi getter per consentire di accedere ai parametri dall'esterno e dei metodi *setter* (o *mutator*) per lo spessore dell'isolante e la C_{sb}, entrambi aggiornati nella fase di fitting. Per questioni relative a `OpenMP`, il setter relativo al parametro σ della prima gaussiana è stato invece implementato all'interno della classe `DosMode1`: infatti, essendo la lista dei parametri condivisa tra tutti i processori (ed essendo le simulazioni eseguite contemporaneamente in parallelo allo step 1 dell'algoritmo), la modifica di σ avrebbe generato un *undefined behavior* (più thread sovrascrivono la stessa variabile in tempi molto vicini e in un ordine imprevedibile).

3.6 Le classi QuadratureRule

Le formule di quadratura descritte nella sezione 2.5 a pagina 20 sono gestite tramite un'interfaccia astratta `QuadratureRule`. Le classi che le utilizzano sono dichiarate `friend` per avere accesso ai nodi e ai pesi calcolati (dichiarati come attributi privati).

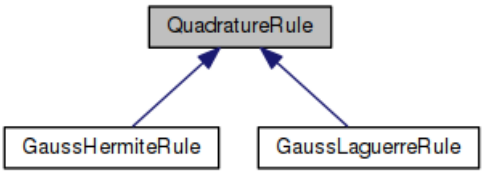

Figura 3.2: Diagramma di ereditarietà della classe QuadratureRule

```
1  class QuadratureRule
2  {
3      public:
4          friend class    GaussianCharge;
5          friend class ExponentialCharge;
6
7          QuadratureRule() = delete;
8          QuadratureRule(const Index &);
9          virtual ~QuadratureRule() = default;
10
11         virtual void apply() = 0;
12         virtual void apply(const GetPot & config) = 0;
13         ...
14
15     protected:
16         Index    nNodes_ ;
17         VectorXr nodes_  ;
18         VectorXr weights_;
19 };
20
21 class GaussHermiteRule : public QuadratureRule
22 {
23     public:
24         ...
25         void apply_iterative_algorithm(const Index &, const Real &);
26         void apply_using_eigendecomposition();
27         ...
28 };
29
30 class GaussLaguerreRule : public QuadratureRule
31 { ... };
```

Il costruttore di default è *deleted*, essendo necessario specificare il numero di nodi di quadratura; sono inoltre disponibili un metodo `apply` che riceve in ingresso oggetto `GetPot` (per la lettura da file di alcuni parametri per il calcolo di pesi e nodi) e un suo equivalente che utilizza invece dei parametri di default. L'interfaccia concreta è stata implementata in `GaussHermiteRule` e `GaussLaguerreRule` che specializzano tali metodi nelle due varianti descritte a livello teorico, cioè un algoritmo diretto basato su una decomposizione agli autovettori/autovalori della matrice di Jacobi e un algoritmo iterativo molto efficiente.

3.7 Le classi Charge

I termini dipendenti da ρ che compaiono nella (2.2) e nella sua discretizzazione (2.3) vengono assemblati attraverso la classe `Charge`. Anch'essa costituisce un'interfaccia astratta, concretizzata sulle diverse relazioni costitutive che possono essere scelte per la DOS, cioè gaussiana (singola o multipla) ed esponenziale.

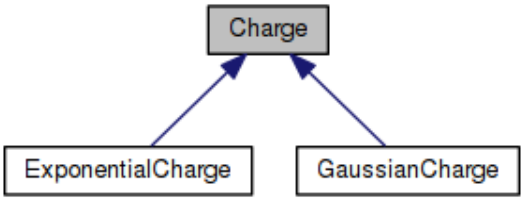

Figura 3.3: Diagramma di ereditarietà della classe Charge

```
1  class Charge
2  {
3      public:
4          Charge() = delete;
5          Charge(const ParamList &, const QuadratureRule &);
6          virtual ~Charge() = default;
7
8          virtual VectorXr  charge(const VectorXr &) const = 0;
9          virtual VectorXr dcharge(const VectorXr &) const = 0;
10
11     protected:
12         const ParamList       & params_;
13         const QuadratureRule & rule_;
14 };
15
16 class GaussianCharge : public Charge
```

```
17 {
18     ...
19     private:
20         Real  n_approx(const Real &, const Real &, const Real &) const;
21         Real dn_approx(const Real &, const Real &, const Real &) const;
22 };
23
24 class ExponentialCharge : public Charge
25 { ... };
```

La classe contiene degli handler a oggetti `ParamList` e `QuadratureRule` (da specificare nel costruttore: anche in questo caso il costruttore di default è *deleted*); grazie al polimorfismo è possibile utilizzare una regola di quadratura arbitraria, purché derivata dalla classe `QuadratureRule`. I metodi `n_approx` e `dn_approx` (che calcolano rispettivamente la concentrazione di elettroni n e la sua derivata rispetto al potenziale φ) non sono presenti nell'interfaccia astratta dal momento che dipendono strettamente dalla relazione costitutiva scelta (nel caso gaussiano, ad esempio, è necessario specificare, oltre al potenziale, anche N_0 e σ mentre per altre forme della DOS i parametri da specificare potrebbero essere diversi).

3.8 Le abstract factory

(a) ChargeFactory **(b)** QuadratureRuleFactory

Figura 3.4: Diagrammi di ereditarietà per le factory

Il **Design Pattern** creazionale **Abstract Factory** è stato utilizzato per permettere di scegliere la relazione costitutiva da utilizzare per la DOS e la regola di quadratura in fase di runtime (in particolare attraverso il file di configurazione `GetPot`). Sono state dunque implementate due factory astratte (rispettivamente per `Charge` e per `QuadratureRule`) senza bisogno di ricompilare la libreria.

```
class ChargeFactory
{
    public:
        ChargeFactory() = default;
        virtual ~ChargeFactory() = default;

        virtual Charge * BuildCharge(const ParamList &, const
            QuadratureRule &) = 0;
};
```

```
class QuadratureRuleFactory
{
    public:
        QuadratureRuleFactory() = default;
        virtual ~QuadratureRuleFactory() = default;

        virtual QuadratureRule * BuildRule(const Index &) = 0;
};
```

Da ciascuna di esse ereditano due factory concrete che costruiscono i prodotti concreti delle due classi. Visto che ogni factory contiene soltanto un *factory method*, sarebbe più semplice utilizzare piuttosto dei Factory Pattern (cioè delle classi contenenti un metodo che, in base ad un'etichetta che ricevono in ingresso, restituiscono una specializzazione concreta del prodotto astratto); tuttavia, si è deciso di mantenere questa struttura più generale che permette una maggiore astrazione in eventuali modifiche future al codice: al momento, infatti, la relazione costitutiva per la DOS e la regola di quadratura sono concetti indipendenti (ad esempio, la regola di Gauss-Hermite non è l'unica[GK96] che permette di approssimare l'integrale (1.9)), ma il codice potrebbe essere adattato per gestire famiglie di prodotti (come le coppie DOS–regola di quadratura) in fase di runtime attraverso l'implementazione di nuovi factory method in `ChargeFactory`. Il client che deciderà quali classi instanziare in funzione dei parametri letti dal file `GetPot` è il metodo `simulate` della classe `DosModel` (che verrà descritta in seguito). Nel caso in cui occorra aggiungere nuove regole di quadratura o nuove forme per la DOS, grazie alla potenza di questo pattern l'interfaccia già presente non dev'essere modificata ma è sufficiente creare le rispettive factory concrete e aggiungere nel client le regole per la gestione di questi nuovi prodotti: ciò fornisce un'importante garanzia di protezione da possibili errori.

3.9 Solutori

La classe astratta `PdeSolver1D` fornisce l'interfaccia per gestire un solutore per equazioni alle derivate parziali nel caso monodimensionale. Anche in questo caso il costruttore di default è *deleted*: occorre infatti, in fase di creazione di un oggetto, specificare la mesh di riferimento (sotto forma di un vettore di reali); sono disponibili metodi per assemblare le matrici di discretizzazione relative a termini di diffusione-trasporto, sola diffusione (matrice di stiffness) o reazione (matrice di massa).

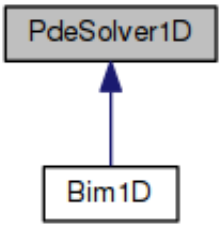

Figura 3.5: Diagramma di ereditarietà della classe PdeSolver1D

```
class PdeSolver1D
{
    public:
        friend class NonLinearPoisson1D;

        PdeSolver1D() = delete;
        PdeSolver1D(VectorXr &);
        virtual ~PdeSolver1D() = default;

        virtual void assembleAdvDiff(...) = 0;
        virtual void assembleStiff  (...) = 0;
        virtual void assembleMass   (...) = 0;
        ...
    protected:
        ...
};

class Bim1D : public PdeSolver1D
{
    public:
        static VectorXr log_mean(const VectorXr &, const VectorXr &);
        static std::pair<VectorXr, VectorXr> bernoulli(const VectorXr &);
};
```

Da `PdeSolver1D` deriva la classe `Bim1D` che implementa il BIM nel caso in esame (la documentazione del codice spiega nel dettaglio quale tipologia di equazioni pos-

sono essere risolte utilizzando questa classe); all'interno di essa sono inoltre definiti dei metodi ausiliari (dichiarati `static` in quanto indipendenti dall'istanziazione di un oggetto `Bim1D`) utilizzati per l'assemblaggio delle matrici (in particolare per calcolare la media logaritmica *element-wise* tra due vettori e per valutare la funzione di Bernoulli in un punto e nel suo opposto utilizzandone le proprietà asintotiche).

```
1  class NonLinearPoisson1D
2  {
3      public:
4          NonLinearPoisson1D() = delete;
5          NonLinearPoisson1D(const PdeSolver1D &, const Index & = 100,
               const Real & = 1.0e-6);
6          virtual ~NonLinearPoisson1D() = default;
7
8          void apply(const VectorXr &, const Charge &);
9          ...
10     private:
11         ...
12 };
```

Infine, la classe `NonLinearPoisson1D` implementa il metodo di Newton descritto nella sezione 2.1 a pagina 13 e utilizza un oggetto di tipo `PdeSolver1D` per la discretizzazione; viene così risolta l'equazione (1.13) nel potenziale, calcolata la carica totale del dispositivo e da ultimo la capacità elettrica. Il costruttore riceve in ingresso un oggetto polimorfico di classe `PdeSolver1D` (di cui verrà salvata una referenza costante in un *handler* privato), il numero massimo di iterazioni e la tolleranza da utilizzare per verificare la convergenza del metodo. Dopo aver costruito l'oggetto, la chiamata al metodo `apply` (al quale occorrerà specificare la mesh, la guess iniziale e la relazione costitutiva per l'assemblaggio il sistema) risolverà l'equazione.

3.10 Il namespace numerics

All'interno del namespace `numerics` vengono implementati gli algoritmi presentati nella sezione 2.6.1 a pagina 22. Tutte le funzioni definite si basano sulle strutture dati della libreria `Eigen`.

```
namespace numerics
{
    template<typename ScalarType>
    VectorX<ScalarType> sort(const VectorX<ScalarType> &);

    template<typename ScalarType>
    VectorXpair<ScalarType> sort_pair(const VectorX<ScalarType> &);
    ...
    Real trapz(const VectorXr &, const VectorXr &);
    VectorXr deriv(const VectorXr &, const VectorXr &);
    Real interp1(...);

    Real error_L2(...);
}
```

L'algoritmo di sort è una funzione template al variare del tipo dello scalare degli elementi del vettore (ad esempio: `double`, `int` etc.). Al suo interno viene richiamato l'algoritmo `std::sort` della Standard Template Library del C++. La funzione `sort_pair` ordina il vettore tenendo traccia delle posizioni e restituisce dunque un vettore di coppie valore–posizione prima del sort; l'algoritmo `std::sort` in questo caso viene utilizzato con la seguente **lambda function** (funzionalità dello standard C++11) che implementa una relazione d'ordine per le coppie:

```
auto sort_pair_lambda = [] (const std::pair<ScalarType, Index> & v1,
                            const std::pair<ScalarType, Index> & v2)
-> bool
{
    return (v1.first < v2.first);
};
```

Inoltre è implementata la regola dei trapezi per approssimare l'integrale di una funzione (discretizzata in un vettore tramite i valori nodali) su un vettore di punti (che verrà utilizzata in particolare nella funzione `error_L2` per calcolare la distanza in norma L^2 tra due funzioni discrete), una regola per approssimare numericamente la derivata di una funzione e infine una formula per interpolare linearmente un vettore (definito su un dominio discreto) su un set di nuovi punti.

3.11 La classe DosModel

È ora possibile, avendo introdotto il funzionamento generale della libreria, descrivere in dettaglio la classe `DosModel` che si occupa di costruire il modello e di risolverlo, salvando i risultati in output. La sua interfaccia è la seguente:

```
class DosModel
{
    public:
        DosModel();
        explicit DosModel(const ParamList &);
        virtual ~DosModel() = default;

        void simulate(...);
        void post_process(...);

        void save_plot(...) const;
        ...

    private:
        bool initialized_;
        ParamList params_;
        Real V_shift_;
        ...
};
```

È bene osservare che la presenza del costruttore di default in questo caso è fondamentale: un oggetto `DosModel` viene creato, durante il loop parallelo di simulazione, da ciascun thread tramite il costruttore di default. Tuttavia, essendo necessario specificare una `ParamList` in ingresso (possibile solo dopo aver letto alcune variabili dal file di configurazione), è stata utilizzata una variabile booleana `initialized_` che di default assume il valore `false` e viene settata a `true` non appena l'oggetto riceve la lista dei parametri richiesta. Se, in fase di simulazione, `imitialized_` non ha il valore `true` viene lanciata una eccezione. Il costruttore che riceve la `ParamList` è dichiarato `explicit` (keyword per informare il compilatore che non può essere interpretato come un *conversion constructor*): in questo modo vengono evitate conversioni di tipo implicite; senza questa precauzione sarebbe infatti possibile inizializzare `DosModel` con un `RowVectorXr`, che è il tipo con cui si costruisce la `ParamList`.

Il metodo `simulate` viene invocato per lanciare una simulazione; riceve in ingresso soltanto un oggetto di classe `GetPot` e alcune stringhe che riguardano, ad esempio, il nome del file contenente i dati sperimentali, la directory in cui salvare gli output etc. Questo metodo, dopo aver aperto i file di output, crea la mesh a partire dalle dimensioni del dispositivo specificate in `ParamList` e inizializza il vettore del range di tensioni V_g applicate, come specificato nel file di input; successivamente, vengono assemblate le matrici del sistema, calcolati nodi e pesi di quadratura (tramite `QuadratureRuleFactory`) e inizializzato un oggetto di tipo `Charge` in base alla relazione costitutiva desiderata (tramite `ChargeFactory`). A questo punto inizia la simulazione vera e propria: per ogni valore di V_g viene invocato il metodo di Newton per la risoluzione dell'equazione e salvati i risultati in delle matrici (le righe indicano la coordinata spaziale e le colonne quella temporale). Al termine del ciclo, attraverso una chiamata al metodo `post_process`, viene effettuato il post-processing descritto nella sezione 2.3 a pagina 18 e infine salvati i risultati della simulazione. In output vengono salvati:

- un file `_info.txt` che contiene informazioni sull'andamento della simulazione (iterazione corrente, tempo di esecuzione etc.) e alcune quantità di interesse (come $V_{\texttt{shift}}$, la coordinata del centro di carica e gli errori rispetto ai dati sperimentali in norma L^2, H^1 e la distanza tra i massimi della derivata);
- un file `_CV.csv`, contenente i valori sperimentali e simulati della capacità $C(V_g)$ e della sua derivata $\frac{dC}{dV_g}(V_g)$;
- un plot di confronto `_plot.png` tra le due curve e, in una sotto-directory, il corrispondente script `Gnuplot` `_plot.gp` utilizzato per generarlo.

3.12 Casi test

Verrà ora descritto il codice relativo ai due casi test, cioè `simulate_dos` per la simulazione in parallelo del modello con diversi parametri in input e `fit_dos` per l'esecuzione parallela del fitting del parametro σ.

3.12.1 SIMULATE_DOS

Le fasi di cui si compone il `main` relativo a questo test sono le seguenti:

1. vengono definiti gli oggetti `GetPot`, per effettuare il parsing della linea comando (da cui è possibile specificare il file di configurazione e la relativa directory) e dei parametri impostati nello stesso file (di default viene cercato il file `../config.pot`);

2. vengono letti dal file di configurazione: i nomi dei file contenenti i dati sperimentali e i parametri del modello (verranno utilizzati per creare un oggetto `ParamList`), l'indice delle simulazioni da effettuare, il numero di thread per l'esecuzione parallela e i nomi della directory in cui salvare l'output del programma;

3. inizia a questo punto il loop parallelo: ogni thread crea un oggetto di classe `DosModel` e, dopo aver importato tramite la classe `CsvParser` la riga rispettiva, viene inizializzata la lista dei parametri;

4. infine, viene chiamato il metodo `simulate` della classe `DosModel` e vengono stampate sullo *standard output* le informazioni relative allo stato delle simulazioni.

3.12.2 FIT_DOS

A differenza del caso precedente, in `fit_dos` il parallelismo non è utilizzato per eseguire in parallelo simulazioni relative a righe diverse nel file di input ma per eseguire il primo step del ciclo di ottimizzazione. Il precedente punto 4 viene dunque sostituito dalla procedura seguente. All'interno di un loop esterno (che termina in un numero di iterazioni fissato dall'utente nel file di configurazione):

1. viene individuato il valore ottimo di σ, eseguendo in parallelo tutte le simulazioni al variare del parametro in un range discreto (anch'esso specificato dall'utente) e identificando il valore per cui l'errore sia minore in una norma scelta tra $L^2(\Omega)$, $H^1(\Omega)$ su $C(V_g)$ oppure la distanza tra i picchi in $\frac{dC}{dV_g}(V_g)$.

2. viene aggiornato dunque il valore di C_{sb}, in funzione della σ individuata;

3. viene aggiornato il valore di t_{semic}.

I file di output vengono salvati con un nome nel formato `i_j_k`, dove `i` è l'indice della simulazione, `j` l'iterazione del loop di ottimizzazione e `k` l'indice della σ attuale all'interno del range. Il file `_fit.txt` contiene le informazioni sul processo di ottimizzazione.

4.1 Risultati delle simulazioni

Il programma è stato eseguito su diversi file di input. Verranno in seguito mostrati i grafici delle simulazioni e dell'analisi di sensitività, in cui si vuole mettere in evidenza la dipendenza delle curve finali da ciascun singolo parametro.

È stato considerato un condensatore MIS di tipo n (per cui vale cioè l'approssimazione che soltanto gli elettroni siano portatori di carica) basato sul polimero Poly{[N,N'-bis(2-octyldodecyl)-1,4,5,8-naphthalene dicarboximide-2,6-diyl]-alt-5,5'-(2,2'-bithiophene)} (P(NDI2OD-T2))[Yan+09]; si tratta di una molecola ad alta mobilità e che offre una buona iniezione di elettroni in presenza di elettrodi in oro (il materiale utilizzato nel dispositivo in esame). La forma considerata per la DOS è gaussiana (singola o doppia). In tabella 4.1 vengono riportati i range di valori utilizzati nelle diverse simulazioni.

Parametro	Valore
t_{semic}	63.57nm
t_{ins}	476nm
ϵ_{semic}	2.9
ϵ_{ins}	2.82
N_0	$10^{24} \div 10^{28} m^{-3}$
σ	$1 \div 10.5\ k_B \cdot 300K$
T	$100 \div 350K$
ϕ_B	$0 \div 2V$

Tabella 4.1: Valori dei parametri utilizzati nelle simulazioni

Le simulazioni sono divise in dieci gruppi, in ciascuno dei quali un parametro viene fatto variare e gli altri mantenuti a un valore fisso, simulando (tranne che nella simulazione 8) un range di tensioni applicate V_g tra $-15V$ e $35V$:

1. singola gaussiana, al variare di σ;
2. singola gaussiana, al variare di N_0;
3. singola gaussiana, al variare di ϕ_B;
4. doppia gaussiana, al variare di σ_2;
5. doppia gaussiana, al variare di $\varphi_{s,2}$;
6. singola gaussiana, al variare del numero di nodi della mesh;
7. singola gaussiana, al variare del numero di step per V_g;
8. singola gaussiana, al variare del range per V_g;
9. singola gaussiana, al variare della temperatura T;
10. doppia gaussiana, al variare della temperatura.

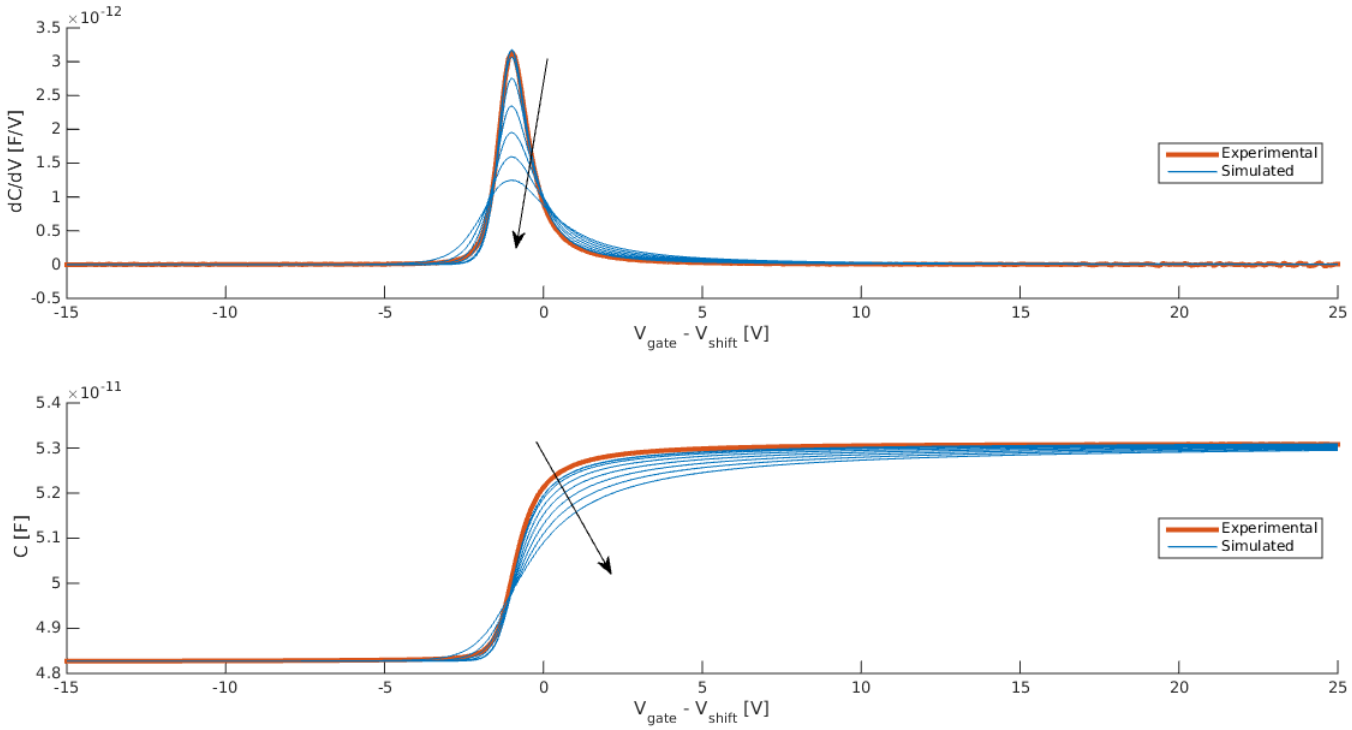

Figura 4.1: Simulazione 1: $N_0 = 10^{27}\,m^{-3}$, σ varia da 1 a 10.5 $k_B \cdot 300K$

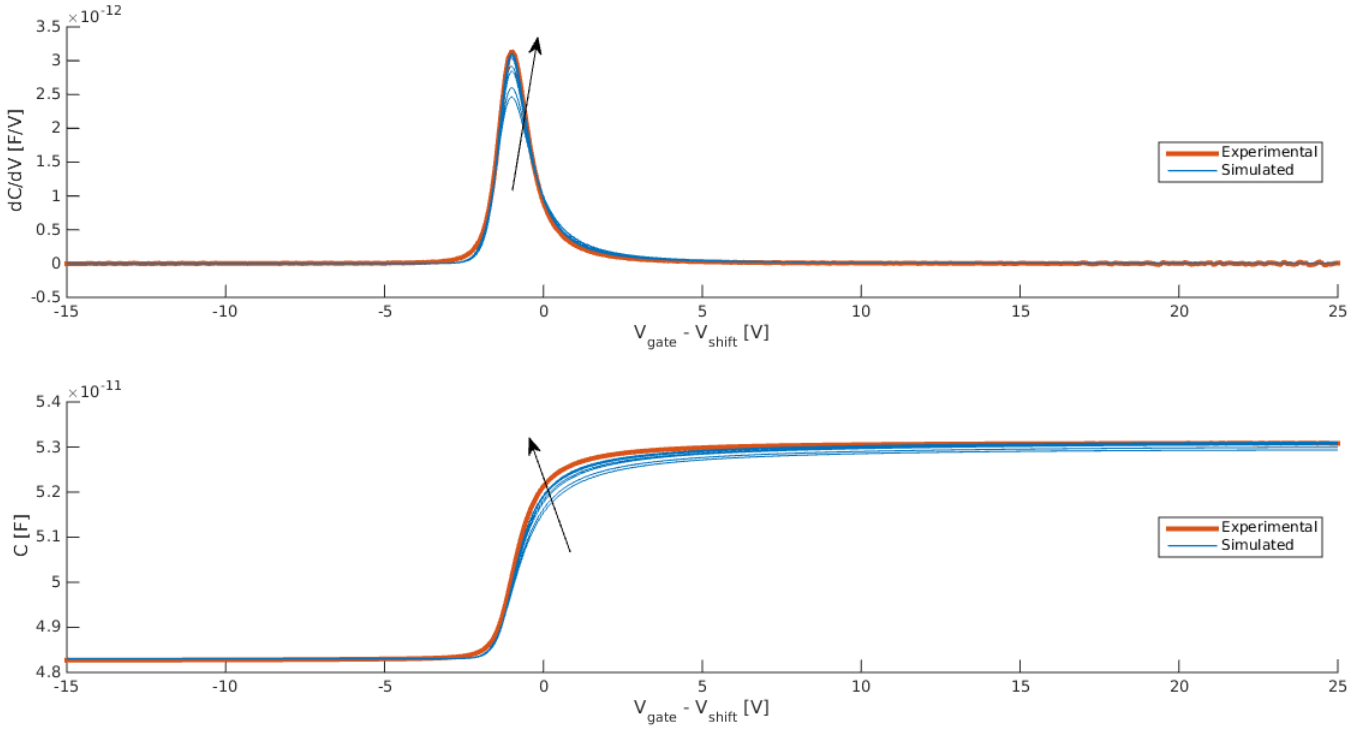

Figura 4.2: Simulazione 2: N_0 varia da 10^{24} a 10^{28}, $\sigma = 3$

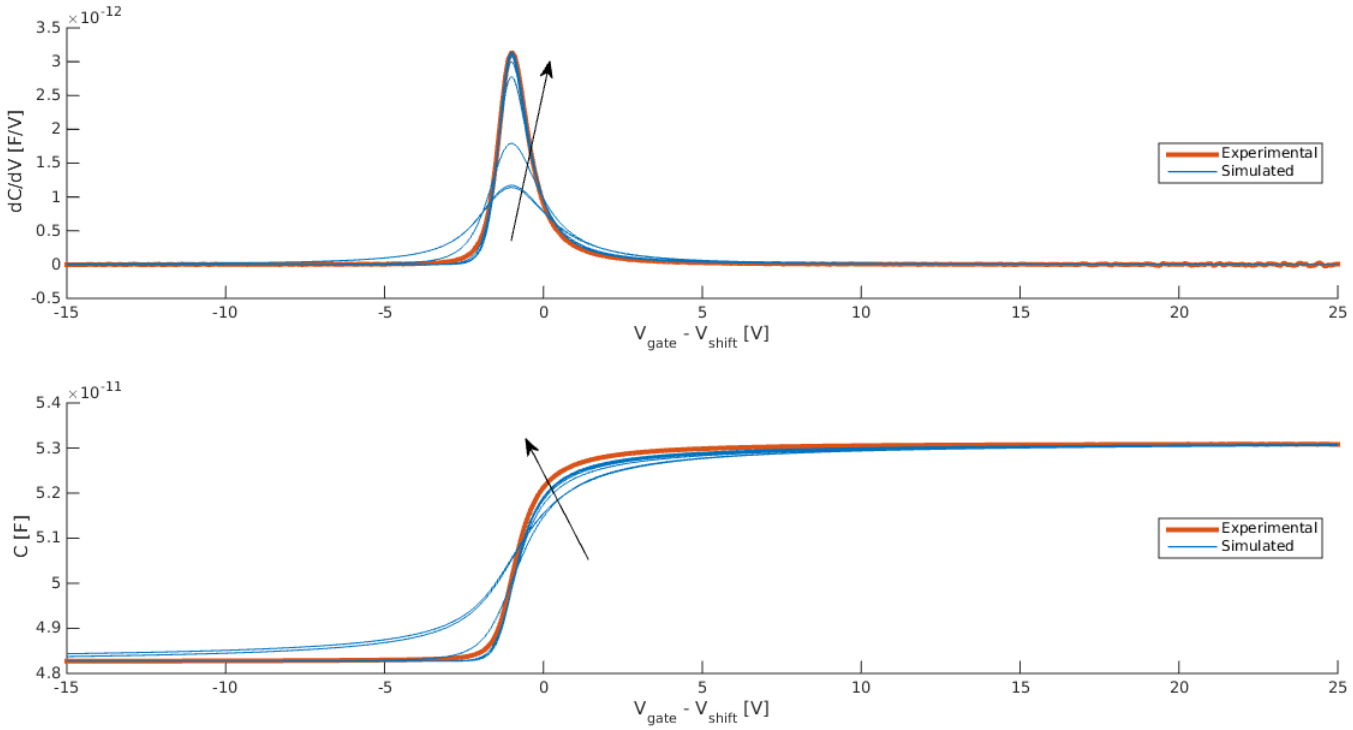

Figura 4.3: Simulazione 3: $N_0 = 10^{27}$, $\sigma = 3$, ϕ_B varia da 0 a 2V

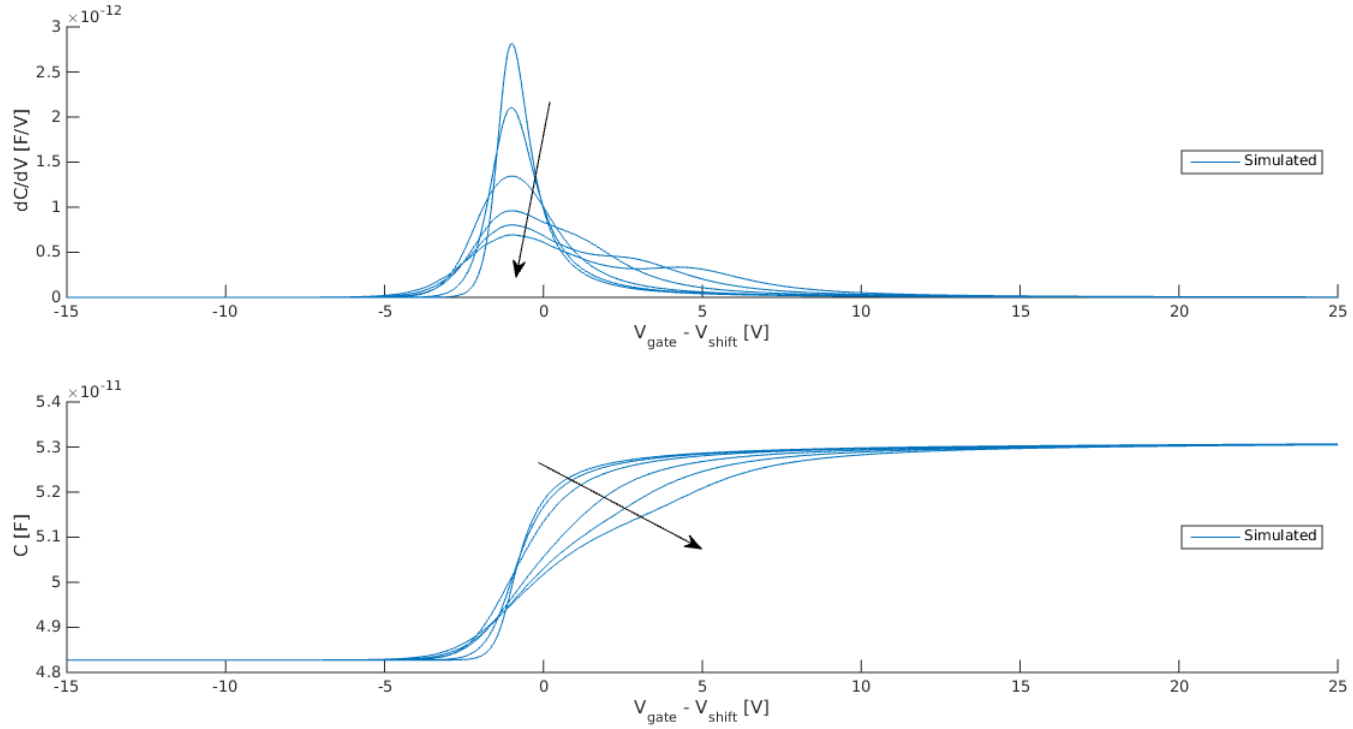

Figura 4.4: Simulazione 4.1: $N_0 = 10^{27}$, $\sigma = 3$, $N_{0,2} = 10^{24}$, σ_2 varia da 4 a 9.5, $\varphi_{s,2} = 0.1$V

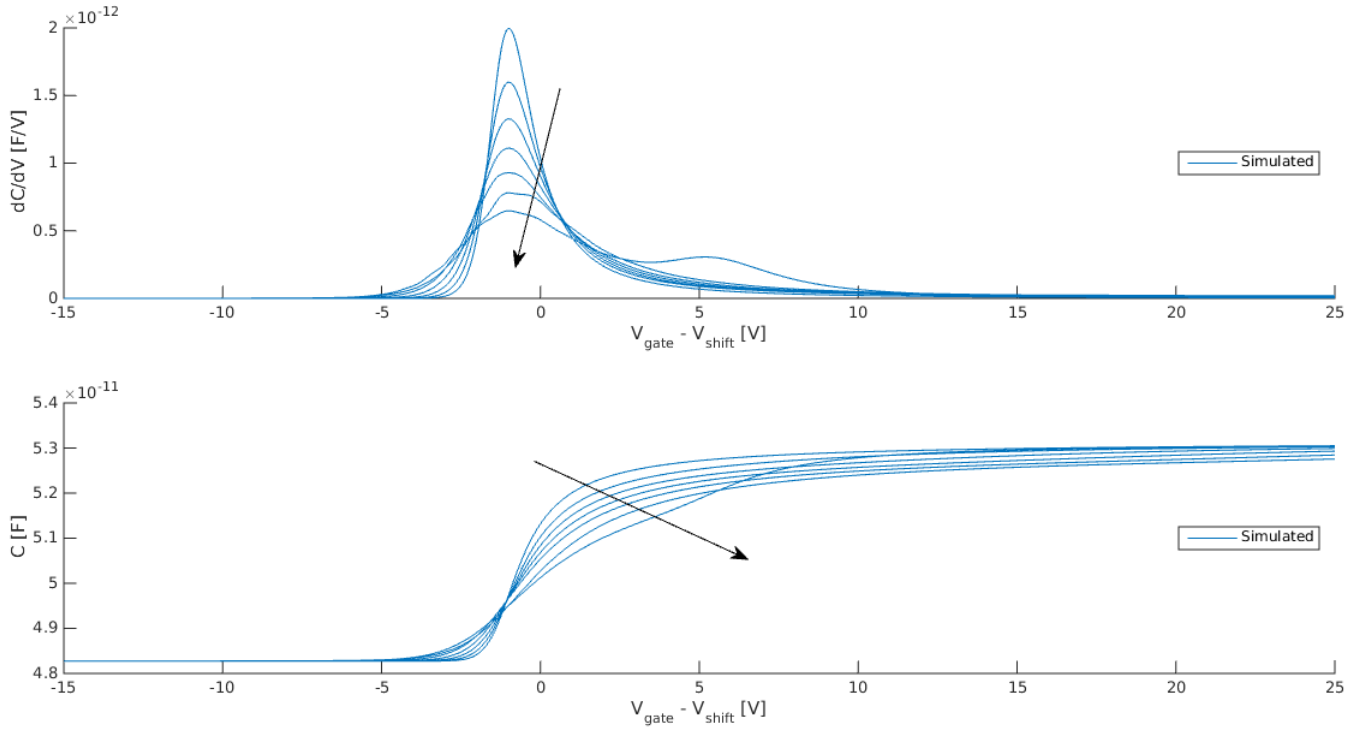

Figura 4.5: Simulazione 4.2: $N_0 = 10^{27}$, $\sigma = 3$, $N_{0,2} = 10^{25}$, σ_2 varia da 4 a 9.5, $\varphi_{s,2} = 0.1$

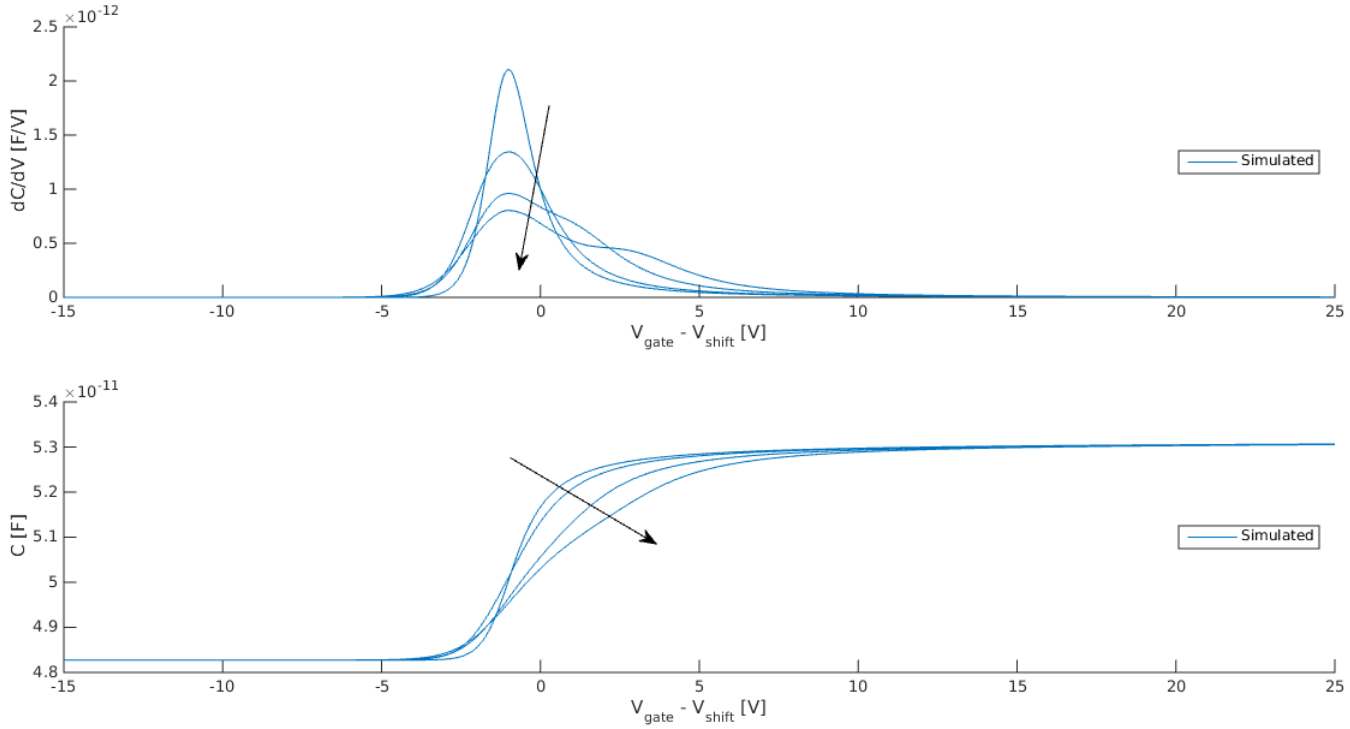

Figura 4.6: Simulazione 5.1: $N_0 = 10^{27}$, $\sigma = 3$, $N_{0,2} = 10^{24}$, σ_2 varia da 5 a 8, $\varphi_{s,2} = 0.1$

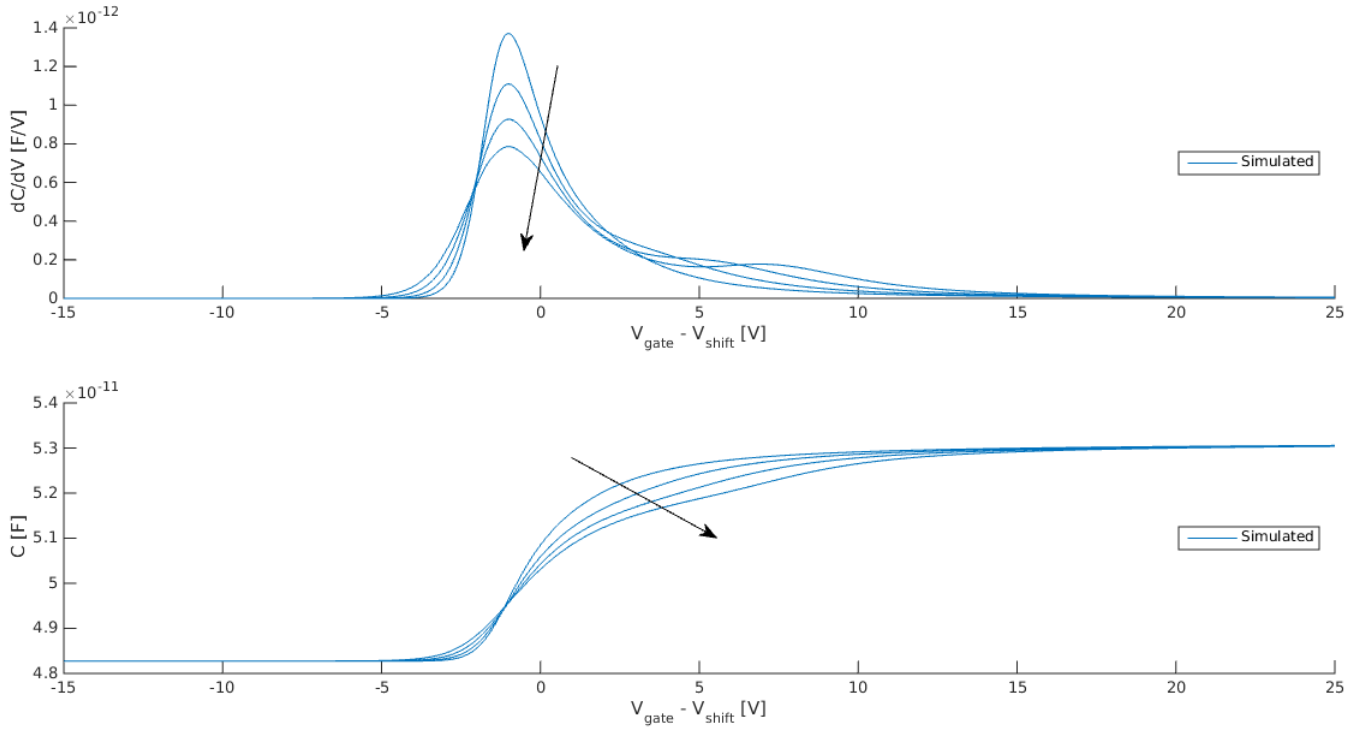

Figura 4.7: Simulazione 5.2: $N_0 = 10^{27}$, $\sigma = 3$, $N_{0,2} = 10^{24}$, σ_2 varia da 5 a 8, $\varphi_{s,2} = 0.2$

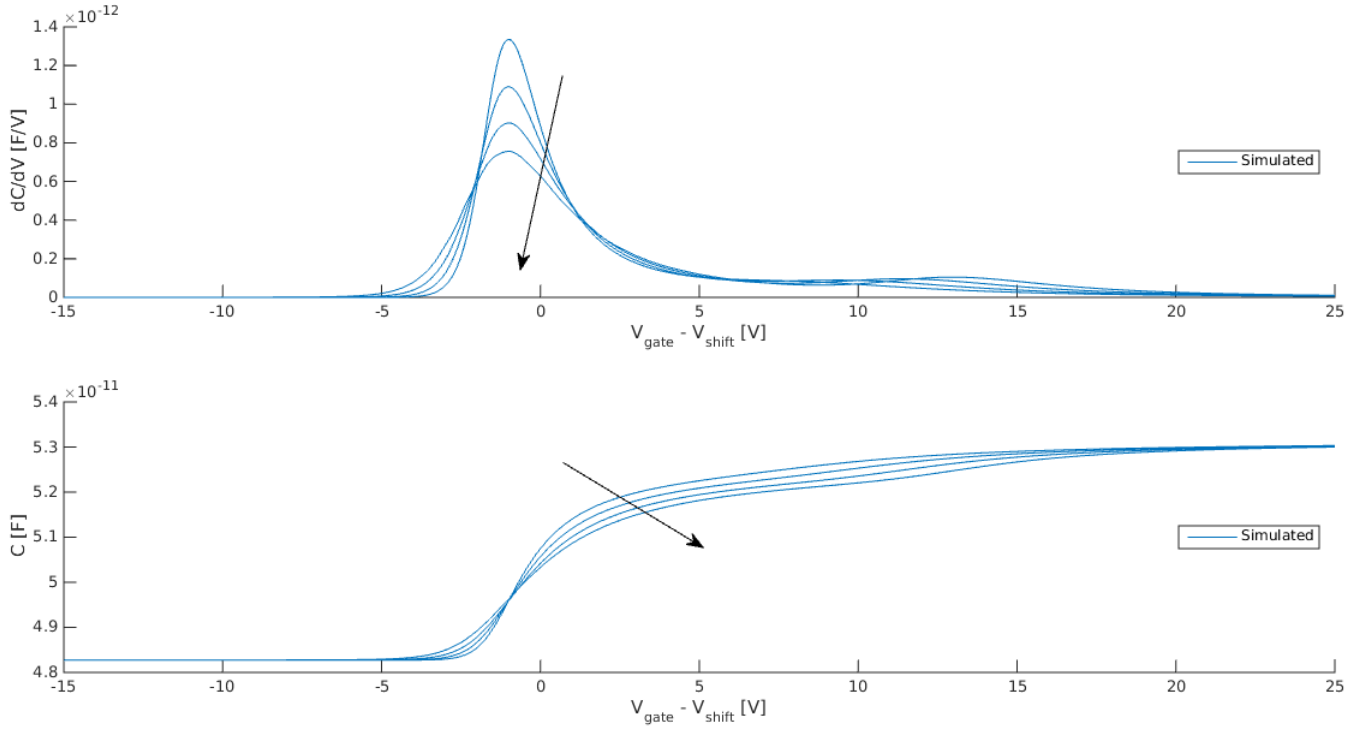

Figura 4.8: Simulazione 5.3: $N_0 = 10^{27}$, $\sigma = 3$, $N_{0,2} = 10^{24}$, σ_2 varia da 5 a 8, $\varphi_{s,2} = 0.3$

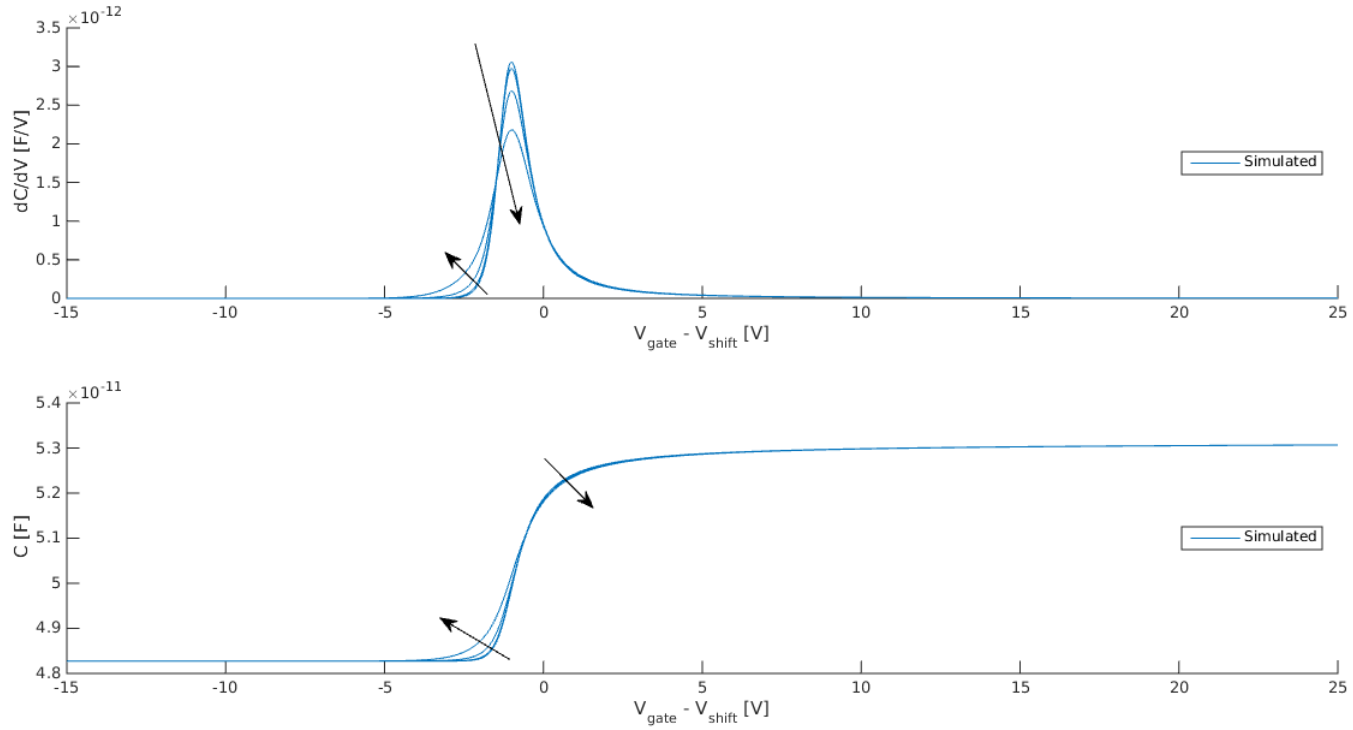

Figura 4.9: Simulazione 5.4: $N_0 = 10^{27}$, $\sigma = 3$, $N_{0,2} = 10^{24}$, σ_2 varia da 5 a 8, $\varphi_{s,2} = -0.1$

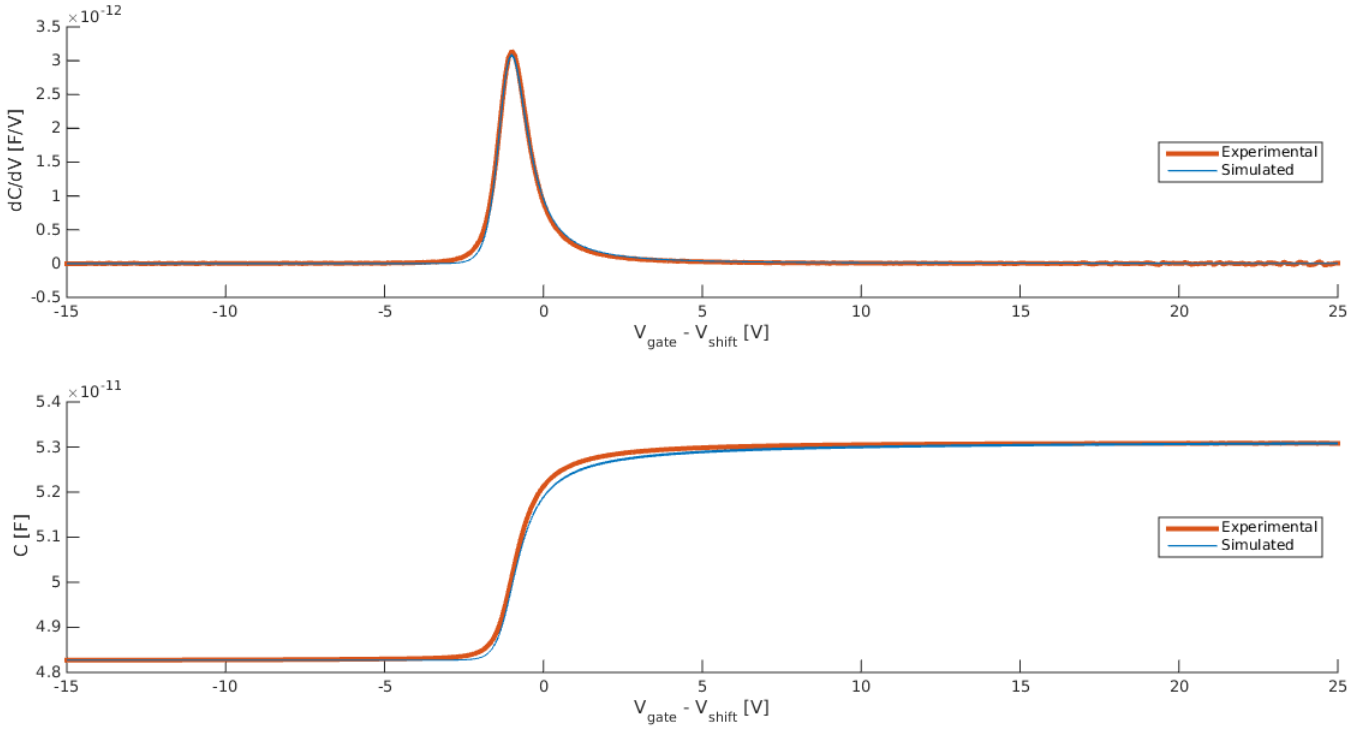

Figura 4.10: Simulazione 6: $N_0 = 10^{27}$, $\sigma = 3$, il no. di nodi della mesh varia da 51 a 1201

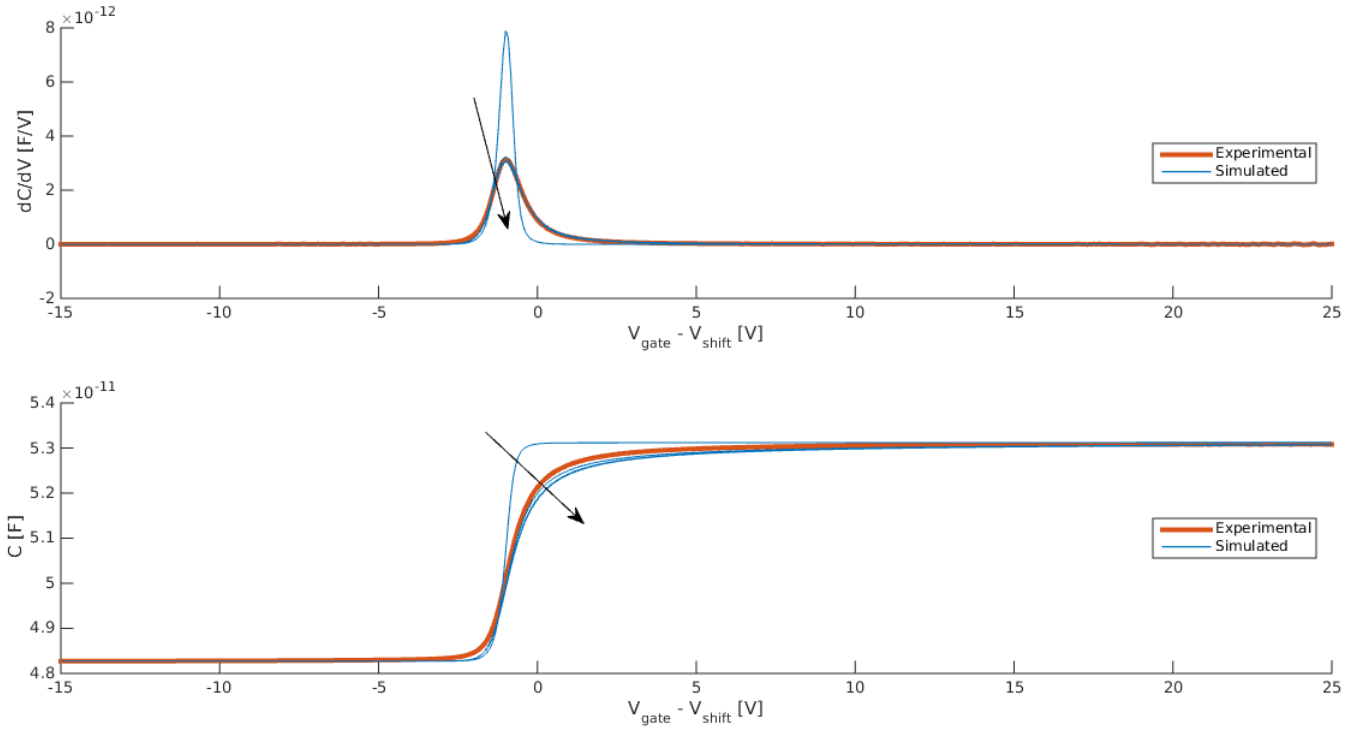

Figura 4.11: Simulazione 7: $N_0 = 10^{27}$, $\sigma = 3$, la dimensione del range varia da 1000 a 8000 step

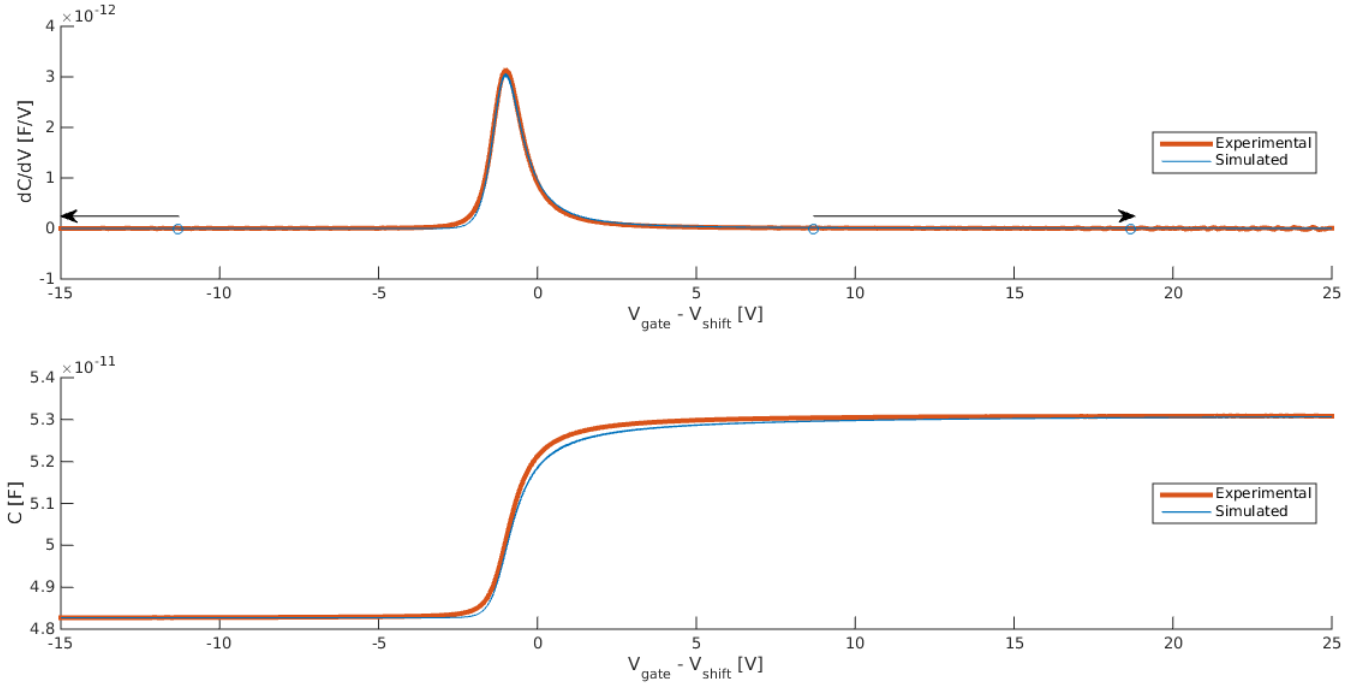

Figura 4.12: Simulazione 8: $N_0 = 10^{27}$, $\sigma = 3$, V_g varia da $[-5,15]$ a $[-55,65]V$

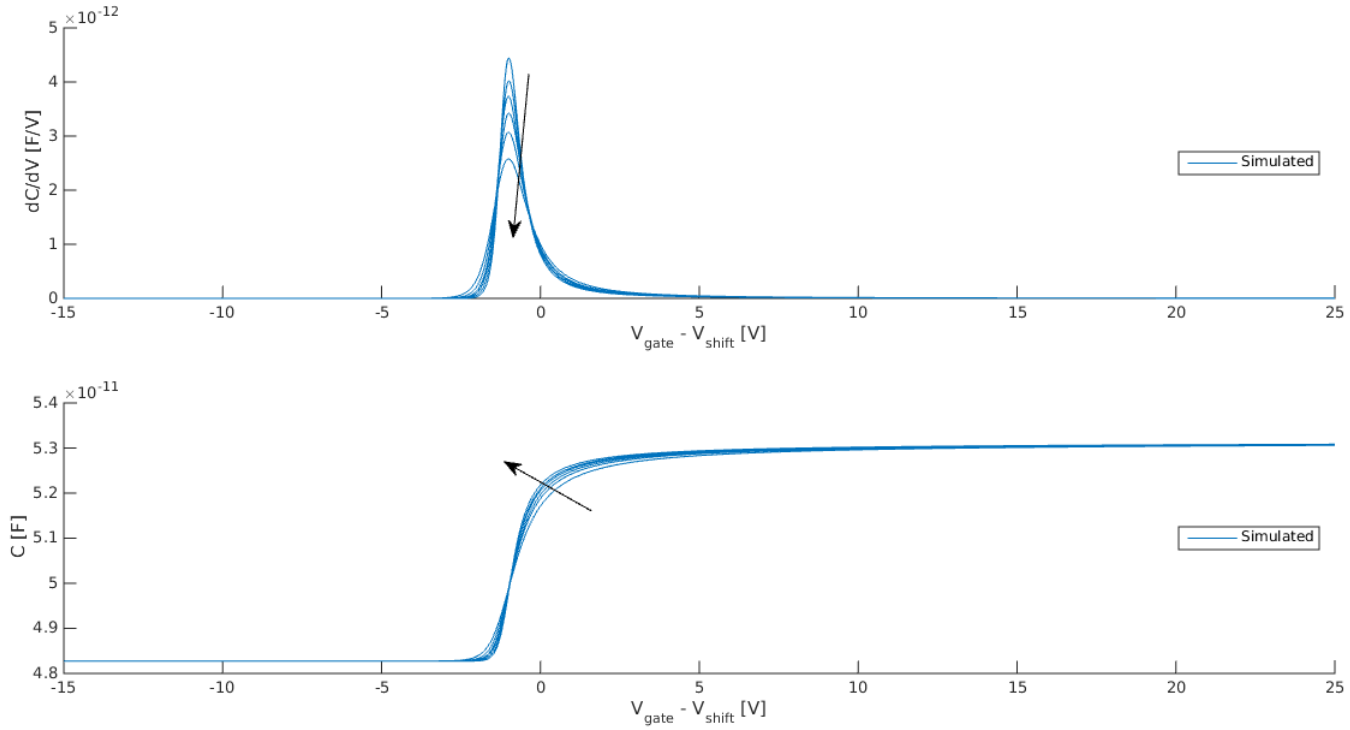

Figura 4.13: Simulazione 9: $N_0 = 10^{27}$, $\sigma = 3$, la temperatura diminuisce da 100 a 350K

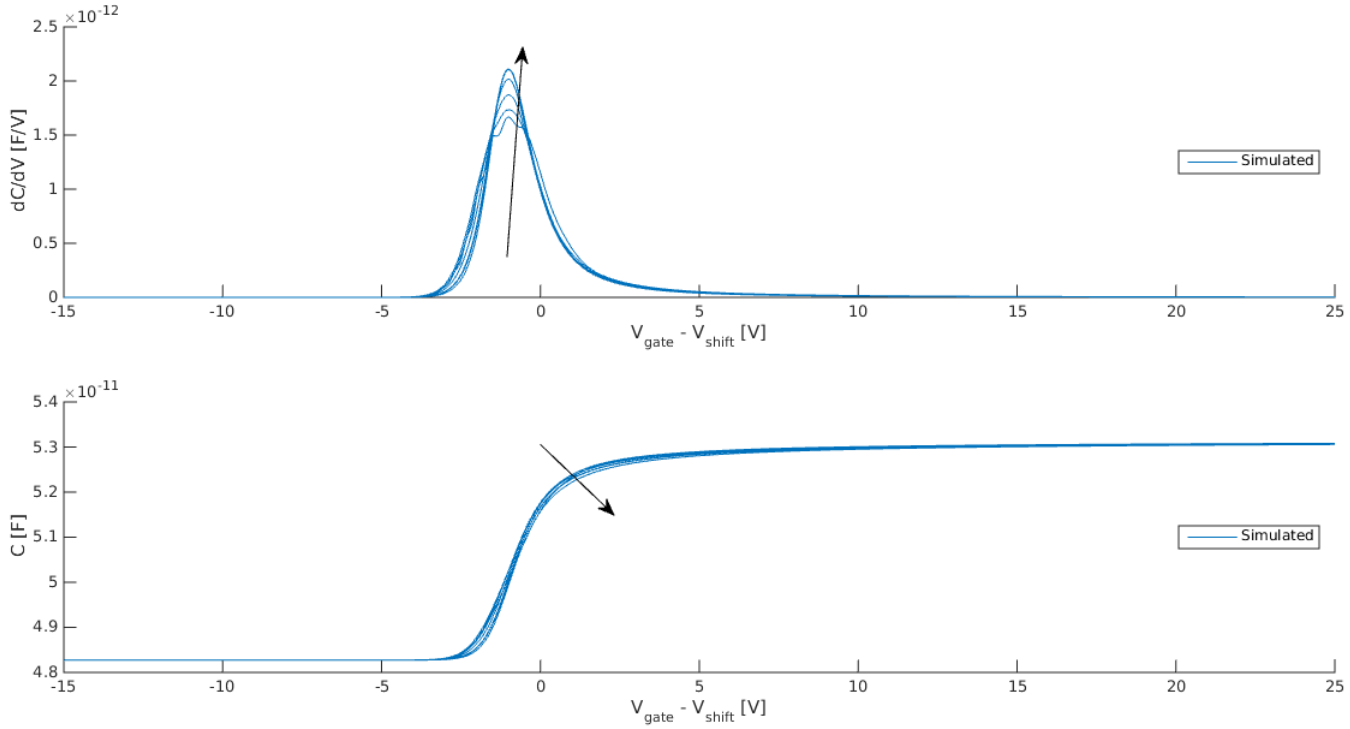

Figura 4.14: Simulazione 10.1: $N_{0,2} = 10^{25}$, $\sigma_2 = 5$, $\varphi_{s,2} = 0.1$, T varia da 100 a 350K

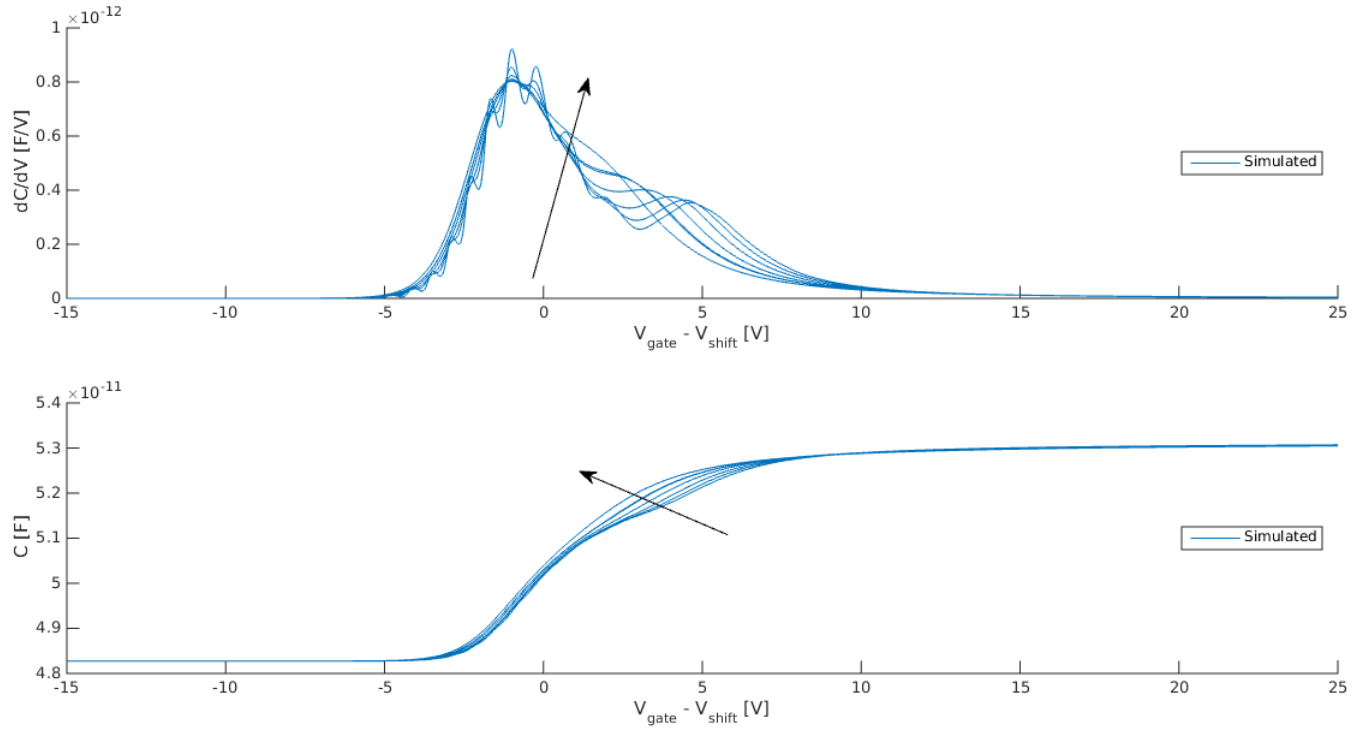

Figura 4.15: Simulazione 10.2: $N_{0,2} = 10^{25}$, $\sigma_2 = 8$, $\varphi_{s,2} = 0.1$, T varia da 100 a 350K

Dai grafici precedenti risulta evidente che tutti i parametri fisici hanno un'importante influenza sulla forma della curva $C(V)$, sia che rientrino direttamente nell'espressione della DOS (N_0, σ etc.) che indirettamente (ϕ_B). In particolare, all'aumentare di σ (che corrisponde, dal punto di vista fisico, ad un maggiore disordine molecolare) la capacità è meno sensibile a variazioni di tensione (la curva è meno pendente) e i risultati della simulazione si allontanano sempre più dalla realtà fisica; si deduce dai grafici che il valore «migliore» di σ sia minore di 3 $k_B \cdot 300K$. N_0 invece ha una minore influenza: una differenza in N_0 di 4 ordini di grandezza non fa variare in maniera così rilevante la forma della curva; il valore stimato e che trova un consenso quasi universalmente riconosciuto in letteratura è di circa $10^{27} m^{-3}$. Il potenziale di barriera ϕ_B (che rientra solo nelle condizioni al contorno della (1.13) all'estremo del semiconduttore) gioca un ruolo opposto a quello di σ: è come se, in presenza di una barriera molto piccola o nulla, il dispositivo risenta meno del potenziale applicato sullo strato isolante e gli effetti in gioco siano dovuti esclusivamente al maggior disordine molecolare, che diventa poco controllabile dall'esterno. Nel caso della doppia gaussiana, all'aumentare di $N_{0,2}$ o σ_2 inizia a formarsi un secondo picco: ciò suggerisce la necessità di adattare il modello a

seconda del caso studiato e in base ai fenomeni fisici di cui si voglia tener conto; i valori più plausibili per il potenziale $\varphi_{s,2}$ sembrano essere negativi (in particolare $-0.1V$ è il valore che genera la forma più simile alle curve sperimentali).
Più interessante è invece il comportamento al variare della temperatura; essa è infatti presente solo all'interno di termini di tipo esponenziale negativo (al denominatore) e questo conduce facilmente a fenomeni di underflow/overflow qualora l'esponente sia, in modulo, molto grande: ciò potrebbe spiegare le cause dell'andamento oscillatorio della soluzione quando la temperatura diminuisce molto.

Infine, i parametri di discretizzazione numerica sembrano garantire una buona robustezza degli algoritmi utilizzati: il numero di nodi della mesh ha un'influenza trascurabile sui risultati finali (permettendo un grande risparmio in termini di oneri computazionali), così come il range simulato di tensioni (le curve si sovrappongono quasi perfettamente in tutti i casi) e la sua dimensione; ad esempio, simulare 2000 oppure 8000 tensioni in ingresso tra -15 e $25V$ non fa variare di molto i risultati (la convergenza rispetto a questo parametro è raggiunta presto, già per un numero di valori pari a 3000).

4.2 Risultati del fitting

L'ottimizzatore `fit_dos` è stato eseguito su alcuni casi notevoli. A partire dai parametri letti dal file di input, vengono effettuate delle simulazioni al variare di σ, che assume valori uniformemente spaziati negli intervalli $[\sigma - \sigma_l, \sigma]$ e $(\sigma, \sigma + \sigma_r]$; in particolare sono stati scelti 3 valori in ciascun sotto-intervallo, $\sigma_l = 2$ e $\sigma_r = 3\ k_B \cdot 300K$.
Il numero di iterazioni del ciclo di ottimizzazione è stato fissato a 3.
Vengono ora riportati i risultati delle varie simulazioni, mostrando per ogni iterazione l'andamento delle quantità d'interesse, cioè il valore ottimale σ_{opt} e i valori aggiornati della capacità parassita C_{sb} e lo spessore del semiconduttore t_{semic}.
In tutti casi i valori iniziali per C_{sb} e t_{semic} sono $1.06695 \cdot 10^{-11}F$ e $63.57269nm$ rispettivamente; inoltre l'algoritmo di ottimizzazione può essere applicato soltanto al caso della gaussiana singola, non essendo previsti step di aggiornamento per eventuali σ_i, $i \geqslant 2$. I risultati ottenuti a partire dalla valutazione della norma H^1 dell'errore, sono i seguenti.

Si nota che in tutte le simulazioni (eccetto che nella 4.4, che non sembra converge-

Iterazione	$\sigma_{opt}[k_B \cdot 300K]$	Errore H^1	$C_{sb}[F]$	$t_{semic}[nm]$
1	1.5	$5.05932 \cdot 10^{-13}$	$1.0667889 \cdot 10^{-11}$	63.5357692
2	0.8	$5.06560 \cdot 10^{-13}$	$1.0665428 \cdot 10^{-11}$	63.4995748
3	0.1	$5.07202 \cdot 10^{-13}$	$1.0664914 \cdot 10^{-11}$	63.4920216

Tabella 4.2: Singola gaussiana con $N_0 = 10^{27} m^{-3}$, $\sigma = 3.5$

Iterazione	$\sigma_{opt}[k_B \cdot 300K]$	Errore H^1	$C_{sb}[F]$	$t_{semic}[nm]$
1	2.5	$5.76331 \cdot 10^{-13}$	$1.0678307 \cdot 10^{-11}$	63.6890038
2	0.5	$4.75813 \cdot 10^{-13}$	$1.0665105 \cdot 10^{-11}$	63.4948321
3	0.1	$5.09179 \cdot 10^{-13}$	$1.0664915 \cdot 10^{-11}$	63.4920301

Tabella 4.3: Singola gaussiana con $N_0 = 10^{27} m^{-3}$, $\sigma = 4.5$

Iterazione	$\sigma_{opt}[k_B \cdot 300K]$	Errore H^1	$C_{sb}[F]$	$t_{semic}[nm]$
1	3	$6.51071 \cdot 10^{-13}$	$1.0686212 \cdot 10^{-11}$	63.8053418
2	1	$4.92887 \cdot 10^{-13}$	$1.0665887 \cdot 10^{-11}$	63.5063261
3	1	$5.33654 \cdot 10^{-13}$	$1.0665845 \cdot 10^{-11}$	63.5057004

Tabella 4.4: Singola gaussiana con $N_0 = 10^{27} m^{-3}$, $\sigma = 5$

Iterazione	$\sigma_{opt}[k_B \cdot 300K]$	Errore H^1	$C_{sb}[F]$	$t_{semic}[nm]$
1	4	$9.67710 \cdot 10^{-13}$	$1.0705384 \cdot 10^{-11}$	64.0876958
2	2	$4.42486 \cdot 10^{-13}$	$1.0672103 \cdot 10^{-11}$	63.5977435
3	0.1	$5.03621 \cdot 10^{-13}$	$1.0664942 \cdot 10^{-11}$	63.4924218

Tabella 4.5: Singola gaussiana con $N_0 = 10^{27} m^{-3}$, $\sigma = 6$

re nelle 3 iterazioni) il trend sia rivolto verso valori di σ_{opt} molto bassi (addirittura 0.1 $k_B \cdot 300K$, che è il limite inferiore imposto nel codice per evitare valori negativi). Questo risultato sembrerebbe in accordo con l'andamento riportato in figura 4.1 ma occorre considerare che $\sigma \to 0$ è una condizione ideale che allontana i risultati dal senso dell'interpretazione fisica (corrisponde infatti alla totale assenza di disordine molecolare); inoltre dai grafici riportati nella sezione precedente emerge che $\sigma = 3$ è un buon valore ad esempio, per il quale la «vicinanza» con la curva sperimentale è soddisfacente in molte configurazioni degli altri parametri. Nonostante questo comportamento anomalo sulla σ, i risultati per C_{sb} e t_{semic} sembrano godere di una maggiore precisione: i valori si assestano attorno a $1.06649 \cdot 10^{-11} F$ e $63.49 nm$ rispettivamente. È dunque lecito a questo punto porre la questione se la norma H^1 sia la più adatta, nella caratterizzazione dell'ottimalità, a descrivere l'errore rispetto ai dati sperimentali. A tal riguardo, le stesse simulazioni sono state ripetute minimizzando invece la distanza tra le ordinate dei picchi nelle curve $\frac{dC}{dV_g}$.

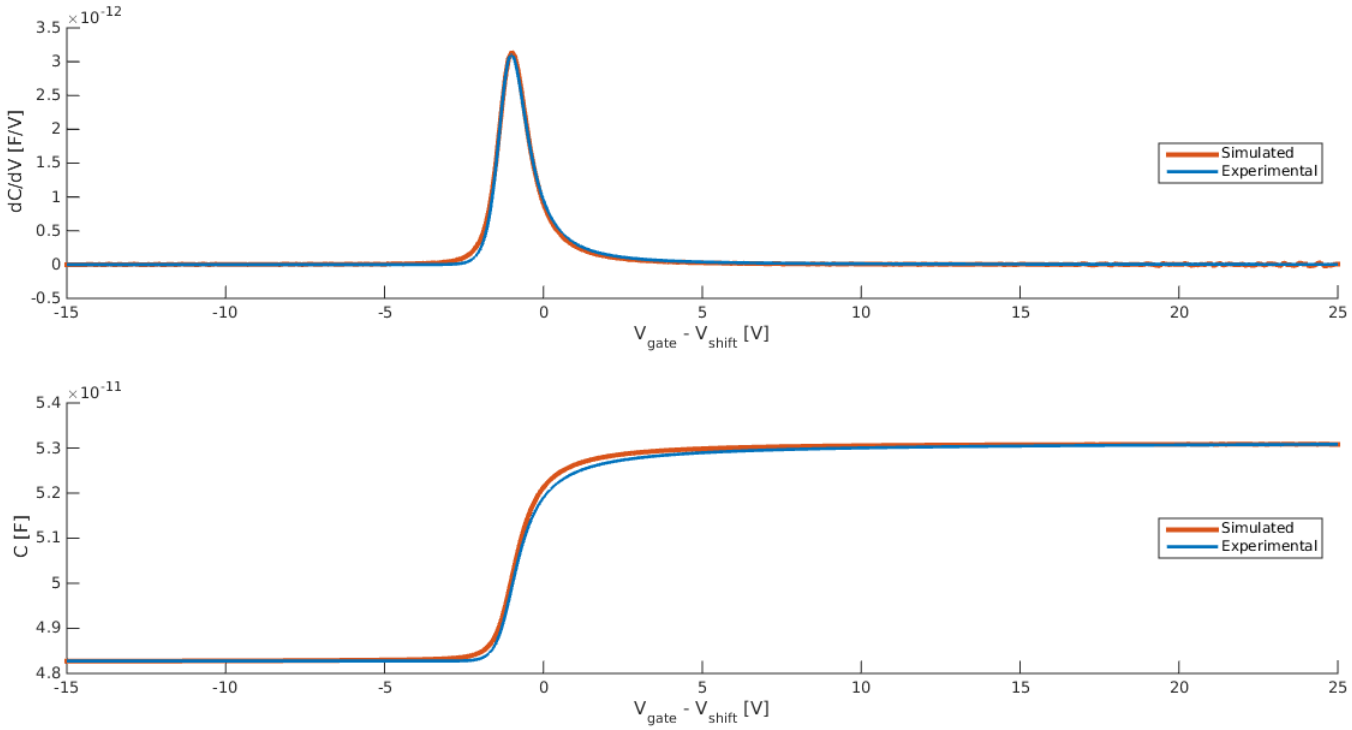

Figura 4.16: Risultati del fitting: confronto per $\sigma = 0.1$

Iterazione	$\sigma_{opt}[k_B \cdot 300K]$	Distanza picchi	$C_{sb}[F]$	$t_{semic}[nm]$
1	1.5	$4.50839 \cdot 10^{-14}$	$1.0667889 \cdot 10^{-11}$	63.5357692
2	0.1	$4.09946 \cdot 10^{-14}$	$1.0664949 \cdot 10^{-11}$	63.4925335
3	0.1	$4.48986 \cdot 10^{-14}$	$1.0664915 \cdot 10^{-11}$	63.4920343

Tabella 4.6: Singola gaussiana con $N_0 = 10^{27} m^{-3}$, $\sigma = 3.5$

Iterazione	$\sigma_{opt}[k_B \cdot 300K]$	Distanza picchi	$C_{sb}[F]$	$t_{semic}[nm]$
1	2.5	$7.57338 \cdot 10^{-14}$	$1.0678307 \cdot 10^{-11}$	63.6890038
2	0.5	$2.85500 \cdot 10^{-14}$	$1.0665105 \cdot 10^{-11}$	63.4948321
3	0.3	$4.45288 \cdot 10^{-14}$	$1.0664983 \cdot 10^{-11}$	63.4930332

Tabella 4.7: Singola gaussiana con $N_0 = 10^{27} m^{-3}$, $\sigma = 4.5$

Iterazione	$\sigma_{opt}[k_B \cdot 300K]$	Distanza picchi	$C_{sb}[F]$	$t_{semic}[nm]$
1	3	$1.40416 \cdot 10^{-13}$	$1.0686212 \cdot 10^{-11}$	63.8053418
2	1	$2.08374 \cdot 10^{-14}$	$1.0665887 \cdot 10^{-11}$	63.5063261
3	0.1	$4.38524 \cdot 10^{-14}$	$1.0664953 \cdot 10^{-11}$	63.4925866

Tabella 4.8: Singola gaussiana con $N_0 = 10^{27} m^{-3}$, $\sigma = 5$

Anche in questo caso, eccetto che nella (4.7), il valore finale di σ_{opt} è 0.1. Tuttavia, l'aver considerato questa diversa misura dell'errore mostra dei risultati che

Iterazione	$\sigma_{opt}[k_B \cdot 300K]$	Distanza picchi	$C_{sb}[F]$	$t_{semic}[nm]$
1	4	$4.35090 \cdot 10^{-13}$	$1.0705384 \cdot 10^{-11}$	64.0876958
2	2	$8.49100 \cdot 10^{-15}$	$1.0672103 \cdot 10^{-11}$	63.5977435
3	0.1	$3.54335 \cdot 10^{-14}$	$1.0664942 \cdot 10^{-11}$	63.4924218

Tabella 4.9: Singola gaussiana con $N_0 = 10^{27} m^{-3}$, $\sigma = 6$

in 3 casi su 4 sono diversi dal caso della norma H^1; in generale sembra che questa seconda strada garantisca una maggiore stabilità, dal momento che i valori finali dei tre parametri sono molto simili per qualsiasi valore iniziale di σ. Ciò conferma la necessità di studiare il problema di ottimizzazione dal punto di vista teorico, per ricavare dei risultati di buona posizione a seconda della norma utilizzata.

4.3 Possibili sviluppi

Questo lavoro si presta ad essere esteso per ampliare le funzionalità della libreria. Il codice è documentato e commentato nei punti essenziali al fine di rendere più agevole questo processo. Innanzitutto si potrebbe rilassare l'approssimazione monodimensionale e considerare una geometria più realistica in **2D** o **3D**: ciò è possibile implementando i rispettivi metodi per l'assemblaggio delle matrici del metodo BIM in dimensione maggiore. Uno studio rilevante riguarderebbe considerare un regime non quasi-statico o addirittura tempo-variante, in cui vengano coinvolti anche fenomeni di trasporto: in questo caso il modello da risolvere sarebbe il sistema drift-diffusion completo presentato nell'introduzione. Un altro percorso consiste nel migliorare l'algoritmo di fitting: formalizzando il problema dal punto di vista teorico è possibile implementare un più generale algoritmo di **ottimizzazione multi-obiettivo** che tenga conto *in toto* della forma della DOS e che ottimizzi a ogni step la σ di ciascuna gaussiana e non solo della prima; inoltre i risultati precedentemente illustrati evidenziano la necessità di un'analisi di buona posizione, in particolare dal punto di vista numerico. Si potrebbe infine tener conto degli effetti dell'adattività di griglia sulla soluzione finale.

4.4 Conclusioni

I risultati ottenuti con il codice sviluppato nel presente progetto sono stati confrontati con le simulazioni eseguite in ambiente `Octave`: la correttezza è garantita

dalla valutazione dell'errore relativo, che non supera mai lo 0.1%; un'analisi più approfondita rivela che questa discrepanza, seppur trascurabile ai fini dell'utilizzo della libreria, deriva totalmente dalla risoluzione di un sistema lineare utilizzato per il calcolo della capacità elettrica (all'interno del metodo `apply` di `NonLinearPoisson1D`): nonostante il metodo utilizzato sia in C++ che nel solutore «backslash» integrato in `Octave` sia lo stesso (basato sulla fattorizzazione di Cholesky in entrambi i casi), sembra esserci una differente implementazione dell'algoritmo tra le `Eigen` e la libreria `Cholmod` (del pacchetto `SuiteSparse`[1]) utilizzata da `Octave`. Infatti, il residuo nei due casi è del tutto paragonabile ma non tutti i termini coincidono esattamente nelle stesse posizioni del vettore. In alcuni casi, questa piccola differenza viene lievemente amplificata nel calcolo della derivata $\frac{dC}{dV_g}$. Tuttavia, pur non potendo stabilire quale tra i due solutori sia più affidabile e considerando che l'entità degli errori è davvero minima su grandezze che si attestano mediamente sull'ordine dei 10^{-13}F, si può affermare che il codice implementato assolve al suo scopo. La scelta del solutore in definitiva adottato merita un breve commento *a posteriori*: una prima versione del codice implementava il metodo iterativo Biconjugate gradient stabilized method (BiCGSTAB), adatto in generale a qualsiasi tipo di matrice quadrata senza particolari richieste sulle sue proprietà, precondizionato con una matrice basata sulla fattorizzazione LU incompleta; successivamente si è deciso di utilizzare il metodo diretto basato sulla decomposizione di Cholesky nella versione generalizzata LDL^T, valido invece solo per matrici simmetriche e definite positive (classe in cui rientrano le matrici gestite nella libreria). Un breve confronto mostra tuttavia che tra le due tecniche le differenze sono pressocché nulle, sia nei risultati che nei tempi di esecuzione (BiCGSTAB è leggermente più oneroso).

In termini di performance, l'esecuzione del codice C++ risulta considerevolmente **più veloce** (essendo un linguaggio compilato) rispetto ad `Octave` (che invece deve essere interpretato); ad esempio, una simulazione che in `Octave` termina in poco più di 30 minuti nella versione C++ è completata in 16 secondi, mentre le simulazioni più onerose (che possono occupare fino a due ore in `Octave`) non superano mai, in C++, gli 8 minuti. Il guadagno è dunque notevole, non solo in termini di tempo direttamente risparmiato ma anche considerato che, ad esempio, eseguire una simulazione in parallelo su molti thread potrebbe richiedere al calcolatore un dispendio di risorse tale da renderlo inutilizzabile fino al termine del processo.

[1] http://faculty.cse.tamu.edu/davis/suitesparse.html

Bibliografia

[BCC98] R. E. Bank, W. M. Coughran e L. C. Cowsar. «The finite volume Scharfetter-Gummel method for steady convection diffusion equations». In: *Computing and Visualization in Science* 1 (4 mar. 1998), pp. 123–136 (cit. a p. 15).

[Cho+14] W. Choi, T. Miyakai, T. Sakurai, A. Saeki, M. Yokoyama e S. Seki. «Non-contact, non-destructive, quantitative probing of interfacial trap sites for charge carrier transport at semiconductor-insulator boundary». In: *Applied Physics Letters* 105.3, 033302 (22 lug. 2014) (cit. a p. 7).

[Coe+05] R. Coehoorn, W. F. Pasveer, P. A. Bobbert e M. A. J. Michels. «Charge-carrier concentration dependence of the hopping mobility in organic materials with Gaussian disorder». In: *Physical Review B* 72, 155206 (15 26 ott. 2005) (cit. a p. vii).

[FRB83] W. Fichtner, D. J. Rose e R. E. Bank. «Semiconductor device simulation». In: *IEEE Transactions on Electron Devices* Vol. 30.9 (9 set. 1983), pp. 1018–1041 (cit. a p. 18).

[FT09] J. A. Freire e C. Tonezer. «Density of states and energetic correlation in disordered molecular systems due to induced dipoles». In: *The Journal of Chemical Physics* 130.13, 134901 (1 apr. 2009) (cit. a p. 7).

[GK96] Alan Genz e B.D. Keister. «Fully symmetric interpolatory rules for multiple integrals over infinite regions with Gaussian weight». In: *Journal of Computational and Applied Mathematics* 71.2 (1996), pp. 299–309 (cit. a p. 40).

[GP05] G. Giustolisi e G. Palumbo. *Introduzione ai dispositivi elettronici*. Milano, Italy: FrancoAngeli, 2005. Cap. 1, 4 (cit. alle pp. 4, 18, 20).

[Hul+04] I. N. Hulea, H. B. Brom, A. J. Houtepen, D. Vanmaekelbergh, J. J. Kelly e E. A. Meulenkamp. «Wide Energy-Window View on the Density of States and Hole Mobility in Poly(p-Phenylene Vinylene)». In: *Physical Review Letters* 93.16, 166601 (15 ott. 2004) (cit. a p. 7).

[Jac99] J. D. Jackson. *Classical Electrodynamics*. 3ª ed. New Jersey, USA: John Wiley & Sons, Inc., 1999 (cit. a p. 1).

[KU98] A. R. Krommer e C. W. Ueberhuber. *Computational integration*. Philadelphia: SIAM, 1998 (cit. a p. 21).

[Kwo+12] S. Kwon, K.-R. Wee, J. W. Kim, C. Pac e S. O. Kang. «Effects of intermolecular interaction on the energy distribution of valance electronic states of a carbazole-based material in amorphous thin films». In: *The Journal of Chemical Physics* 136.20, 204706 (2012-05-29) (cit. a p. 7).

[Liu+11] F. Liu, L. Jack, M. Takaaki e I. Mitsumasa. «Modeling carrier transport and electric field evolution in Gaussian disordered organic field-effect transistors». In: *Journal of Applied Physics* 109.10, 104512 (15 mag. 2011) (cit. a p. viii).

[Mar+09] N. G. Martinelli, M. Savini, L. Muccioli, Y. Olivier, F. Castet, C. Zannoni, D. Beljonne e J. Cornil. «Modeling Polymer Dielectric/Pentacene Interfaces: On the Role of Electrostatic Energy Disorder on Charge Carrier Mobility». In: *Advanced Functional Materials* 19.20 (2009), pp. 3254–3261 (cit. a p. 7).

[OHB12] J. O. Oelerich, D. Huemmer e S. D. Baranovskii. «How to Find Out the Density of States in Disordered Organic Semiconductors». In: *Physical Review Letters* 108.22, 226403 (29 mag. 2012) (cit. a p. 9).

[Pas+05] W. F. Pasveer, J. Cottaar, C. Tanase, R. Coehoorn, P. A. Bobbert, P. W. M. Blom, D. M. de Leeuw e M. A. J. Michels. «Unified Description of Charge-Carrier Mobilities in Disordered Semiconducting Polymers». In: *Physical Review Letters* 94, 206601 (20 23 mag. 2005) (cit. a p. vii).

[Poe+13] C. Poelking, E. Cho, A. Malafeev, V. Ivanov, K. Kremer, C. Risko, J. Brédas e D. Andrienko. «Characterization of Charge-Carrier Transport in Semicrystalline Polymers: Electronic Couplings, Site Energies, and Charge-Carrier Dynamics in PBTTT». In: *The Journal of Physical Chemistry C* 117.4 (2013), pp. 1633–1640 (cit. a p. 7).

[Pre+07] W. H. Press, S. A. Teukolsky, W. T. Vetterling e B. P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. 3ª ed. New York, NY, USA: Cambridge University Press, 2007 (cit. a p. 22).

[Qua08] A. Quarteroni. *Modellistica numerica per problemi differenziali*. 4a ed. Milano, Italy: Springer-Verlag Italia, 2008. Cap. 5 (cit. a p. 15).

[RE11] M. Raja e B. Eccleston. «The significance of Debye length in disordered doped organic devices». In: *Journal of Applied Physics* 110.11, 114524 (13 dic. 2011) (cit. a p. 7).

[Riv+11] J. Rivnay, R. Noriega, J. E. Northrup, R. J. Kline, M. F. Toney e A. Salleo. «Structural origin of gap states in semicrystalline polymers and the implications for charge transport». In: *Physical Review B* 83.12, 121306 (16 mar. 2011) (cit. a p. 7).

[SG69] D. L. Scharfetter e H. K. Gummel. «Large-signal analysis of a silicon Read diode oscillator». In: *IEEE Transactions on Electron Devices* 16.1 (1 gen. 1969), pp. 64–77 (cit. a p. 18).

[SM99] J. C. Scott e G. G. Malliaras. «Charge injection and recombination at the metal-organic interface». In: *Chemical Physics Letters* 229 (1 giu. 1999), pp. 115–119 (cit. a p. 11).

[Sac] Prof. R. Sacco. *Appunti e dispense dal corso di Elettronica computazionale*. URL: http://www1.mate.polimi.it/CN/MNME/metodi_me.php3 (cit. alle pp. 4, 18).

[Sel84] S. Selberherr. *Analysis and simulation of semiconductor devices*. 1st ed. Wien-New York: Springer-Verlag, 1984 (cit. a p. 10).

[Sha99] H. Shao. «Numerical analysis of meshing and discretization for anisotropic convection-diffusion equations with applications». Ph. D. thesis. Department of Computer Science, Duke University, ago. 1999 (cit. a p. 16).

[Sze81] S. M. Sze. *Physics of semiconductor devices*. 2nd ed. New York: John Wiley & Sons, 1981 (cit. a p. 11).

[TM11] A. K. Tripathi e Y. N. Mohapatra. «Correlation of photocurrent and electroabsorption spectra and their temperature dependence for conjugated light emitting polymers: The origin of the corresponding density of states». In: *Physical Review B* 84.20, 205213 (18 nov. 2011) (cit. a p. 7).

[VW09] N. Vukmirović e L.-W. Wang. «Charge Carrier Motion in Disordered Conjugated Polymers: A Multiscale Ab Initio Study». In: *Nano Letters* 9.12 (13 nov. 2009), pp. 3996–4000 (cit. a p. 7).

[VW11] N. Vukmirović e L.-W. Wang. «Density of States and Wave Function Localization in Disordered Conjugated Polymers: A Large Scale Computational Study». In: *The Journal of Physical Chemistry B* 115.8 (3 feb. 2011), pp. 1792–1797 (cit. a p. 7).

[Vri+13] R. J. de Vries, A. Badinski, R. A. J. Janssen e R. Coehoorn. «Extraction of the materials parameters that determine the mobility in disordered organic semiconductors from the current-voltage characteristics: Accuracy and limitations». In: *Journal of Applied Physics* 113.11, 114505 (19 mar. 2013) (cit. a p. 7).

[Yan+09] H. Yan, Z. Chen, Y. Zheng, C. Newman, J. R. Quinn, F. Dötz, M. Kastler e A. Facchetti. «A high-mobility electron-transporting polymer for printed transistors». In: *Nature* 457.7230 (2009), pp. 679–686 (cit. a p. 47).