POLITECNICO DI MILANO

Corso di Studi di Ingegneria Matematica
Orientamento Calcolo Scientifico

Una metodologia XFEM per problemi ellittici 3D con superfici di discontinuità

Relatore: Prof. Luca Formaggia
Correlatore: Dott. Alessio Fumagalli

Guido Francesco Iori
Matr.735708

ANNO ACCADEMICO 2010-2011

Ai miei genitori, a Lucia e a Tommaso

Sommario

Nel campo della simulazione numerica di fenomeni caratterizzati da brusche variazioni nei parametri del modello, è stata sviluppata negli ultimi anni una nuova famiglia di metodi, chiamati XFEM (Extended Finite Element Method), che garantiscono una certa accuratezza nella risoluzione di problemi in cui la soluzione non è regolare.
Questa tesi si pone come obiettivo l'implementazione e la validazione di un metodo XFEM per la risoluzione di un problema ellittico su un dominio tridimensionale attraversato da un *level set* attraverso il quale si impone un salto nella derivata conormale della soluzione. Il metodo implementato è stato proposto da Hansbo e Hansbo nel 2002.
Vengono mostrati gli algoritmi necessari a gestire una griglia attraversata da un numero arbitrario di *level set* con il vincolo che al massimo due di essi attraversino uno stesso elemento. Sono poi spiegate le tecniche di integrazione che permettono di risolvere un problema secondo l'approccio XFEM considerato. Infine, i principali risultati teorici vengono verificati e si analizzano le performance del codice.

Abstract

In numerical simulation of discontinuous phenomena, a new class of methods has been recently developed. These methods, named XFEM (Extended Finite Methods), allow for discontinuities, internal to the element of the mesh, in the approximation across the interface.
The aim of this master thesis is the implementation and validation of an XFEM method to solve an elliptic interface problem in a 3D domain. The method has been proposed by Hansbo and Hansbo in 2002.
Details about the implementation are given, from the geometrical handling of a bidimensional surface cutting the mesh to the numerical techniques of approximation of integrals in the context of the XFEM approach. In the last part of this work, theoretical estimates about convergence of the method are compared with numerical results and the performance of the code is analysed.

Indice

Introduzione 1

1 Il metodo degli Elementi Finiti Estesi 5

1.1 Un'introduzione al metodo . . . 5
1.2 Formulazione del problema . . . 7
1.3 Il problema discreto . . . 8
1.4 Analisi . . . 10
1.4.1 Proprietà di approssimazione dello spazio discreto V^h . . . 11
1.4.2 Stima a priori . . . 12
1.4.3 Stima a posteriori . . . 12
1.5 Cenni sull'implementazione . . . 13

2 Gestione di superfici in griglie 3D 14

2.1 Discontinuità attraverso il dominio . . . 14
2.2 Superfici bidimensionali e funzioni *level set* . . . 16
2.2.1 La classe **AnalyticSurface** . . . 16
2.2.2 La classe **SurfaceSet** . . . 17
2.3 Informazioni sul taglio degli elementi . . . 18
2.3.1 La classe **Marker** e classi figlie . . . 18
2.3.2 La classe **XfemCutElementData** . . . 21
2.4 Algoritmi di taglio: la classe **XfemMeshHandler** . . . 21
2.4.1 Il contenuto della classe . . . 21
2.4.2 Il taglio degli elementi . . . 23
2.4.3 Il taglio delle facce . . . 30
2.4.4 Definizione della regione di appartenenza . . . 31

3 Integrazione del problema differenziale 33

3.1 Le strutture dati presenti in `LifeV` . . . 33
3.2 Parallelismo e partizionamento . . . 34
3.3 Assegnazione dei gradi di libertà . . . 35
3.3.1 Implementazione: le classi **MultipleDOFHandler** e **DOFXfem** . . . 37
3.4 Risoluzione del problema discreto . . . 40
3.4.1 Integrazione su un elemento tagliato . . . 40

3.4.2 Costruzione del sistema lineare: la classe **XfemAssembler** 41
3.4.3 Imposizione delle condizioni al bordo 45
3.5 Visualizzazione dei risultati 45
3.5.1 Esportazione della *mesh* tagliata 46
3.5.2 Interpolazione della soluzione sulla *mesh* tagliata 46

4 Risultati 50
4.1 Verifica dei risultati teorici 50
4.1.1 Ordine di convergenza 51
4.1.2 Analisi degli autovalori 51
4.2 Performace del codice 55
4.2.1 Scalabilità 55
4.2.2 *Load balancing* 56
4.3 Taglio multiplo 57

Conclusioni 60

Bibliografia 62

Ringraziamenti 64

Elenco delle figure

1 Frattura tramite calcolo XFEM (Fonte: `http://www.matthewpais.com/`). . . . 1
2 Pressione in un mezzo poroso fratturato (Fonte: [7], p.28). 2

1.1 Dominio e superficie di discontinuità 7
1.2 *Split* di un elemento attraversato da Γ 9
1.3 Elementi attraversati da Γ per cui non sono valide le assunzioni fatte 11

2.1 *Level set* in 2D 14
2.2 Taglio di un elemento dato un *level set* 24
2.3 Taglio di un elemento dati due *level set* 28
2.4 Elemento tagliato in cui le facce sono ritriangolate in modo non conforme . . . 29

3.1 Confronto tra partizionamento pesato e non pesato 35
3.2 Esempio di *mesh* partizionata 36

4.1 Soluzione approssimata del caso test 52
4.2 Andamento della norma L_2 dell'errore per metodo XFEM e *unfitted* FEM . . . 53
4.3 *Pattern* di sparsità della matrice del test 54
4.4 Decomposizione del dominio per il calcolo a 64 processori 56
4.5 Scalabilità del codice implementato. 56
4.6 Soluzione del test con due *level set* intersecanti il dominio 58

Elenco delle tabelle

2.1 Posizione di un elemento a partire dal segno della funzione *level set* nei vertici. 25

2.2 Assegnazione della *flag* **M_region** di un elemento in una *mesh* attraversata da due superfici. 31

4.1 Confronto tra le norme L_2 dell'errore per XFEM e *unfitted* FEM 51

4.2 Autovalore massimo e minimo al variare della posizione del *level set* 53

4.3 Autovalore massimo e minimo al variare di $h = 1/N$ 54

4.4 Tempi di esecuzione del calcolo su n processori 55

4.5 Confronto dei tempi di esecuzione del calcolo su due processori tra partizionamento pesato e non pesato 57

4.6 Norma L_2 dell'errore in funzione del passo h per il test con due *level set* 59

Introduzione

Un fenomeno discontinuo è caratterizzato da una variazione brusca, nello spazio o nel tempo, di una grandezza fisica. In natura, questo tipo di fenomeni sono molto comuni pertanto è di grande interesse sviluppare metodi di approssimazione numerica che siano in grado di simularne l'evoluzione con sufficiente accuratezza.
Si pensi, ad esempio, a problemi di elastodinamica, come la propagazione di fratture in un solido o il movimento di dislocazioni in un cristallo (vedi figura 1).
Una miscela eterogenea di due o più componenti costituisce un sistema multifase in cui si forma un'interfaccia tra le diverse fasi. Questo è un altro esempio di sistema fisico in cui è presente una discontinuità in alcuni parametri, in questo caso le proprietà fisico-chimiche delle fasi. La simulazione di un flusso multifase richiede delle tecniche che siano in grado di ricostruire o tracciare numericamente l'interfaccia di separazione tra i fluidi non miscibili.

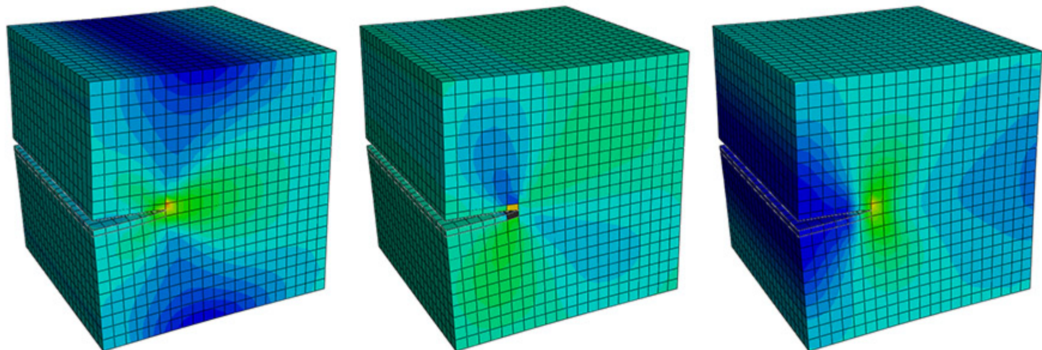


Figura 1: Frattura tramite calcolo XFEM (Fonte: `http://www.matthewpais.com/`).

Anche la simulazione della migrazione di idrocarburi in un bacino sedimentario rientra in questo tipo di fenomeni (figura 2). Un pozzo petrolifero è un mezzo poroso, solitamente eterogeneo, le cui proprietà variano a seconda della posizione. Consiste infatti in un insieme di blocchi di materiale poroso, le cui proprietà di permeabilità possono variare notevolmente, e una rete di fratture, che possono fungere da barriere o canali preferenziali per il flusso dell'idrocarburo. Pertanto anche nel campo della simulazione di flussi in mezzi porosi fratturati, in cui rientra un problema di grande attualità come il sequestro della CO_2, è importante disporre di strumenti numerici adeguati.

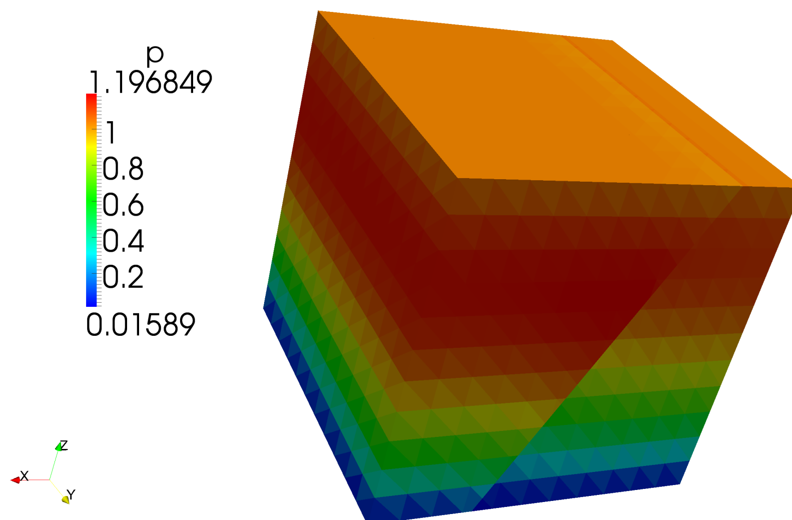


Figura 2: Pressione in un mezzo poroso fratturato (Fonte: [7], p.28).

Diverse tecniche sono state proposte negli ultimi anni nell'ambito degli Elementi Finiti (FEM) per gestire le discontinuità presenti nei modelli fisici ([3],[1]).
Una possibilità è quella di risolvere il problema differenziale su una *mesh* costruita in modo tale che la superficie di discontinuità coincida o sia ben approssimata da un insieme di lati (o facce in 3D) della triangolazione. I risultati ottenuti con questa tecnica sono soddisfacenti ma la costruzione di queste griglie può essere estremamente complessa, soprattutto nel caso 3D e in problemi in cui la superficie di discontinuità cambia nel tempo (è questo il caso della propagazione di una frattura).
Un'altra possibilità è quella di risolvere il problema su una *mesh unfitted* imponendo le condizioni attraverso l'interfaccia tramite un metodo di penalità. Il fatto di risolvere il problema su griglie *unfitted* permette di fare analisi statistiche su diversi scenari di fratture senza essere costretti a costruire una *mesh* per ogni simulazione.
Recentemente sono stati sviluppati i metodi degli Elementi Finiti Estesi (XFEM) o Generalizzati (GFEM) [11] a partire dal metodo degli Elementi Finiti standard [20], ed essi rappresentano validi strumenti per l'analisi di problemi caratterizzati da discontinuità e geometrie complesse. Essi operano su *mesh unfitted* evitando così le complicazioni legate alla creazione di una triangolazione adattata all'interfaccia, tuttavia rispetto al metodo di penalità a cui si è fatto cenno, l'ordine di convergenza è superiore. Il grande vantaggio di questi metodi è la semplificazione della modellazione di fenomeni discontinui tramite un arricchimento locale dello spazio di approssimazione. La discretizzazione del problema si discosterà dunque da quella del metodo FEM standard per quanto riguarda gli elementi attraversati dall'interfaccia.

Questo lavoro di tesi ha come obiettivo l'implementazione del metodo XFEM proposto da Hansbo e Hansbo [14] nel 2002 per risolvere un' equazione ellittica su un dominio 3D attraversato da superfici di discontinuità.
L'idea che sta alla base del metodo è quella di raddoppiare i gradi di libertà degli elementi attraversati dall'interfaccia. In questo modo, è possibile rappresentare indipendentemente la

soluzione da un lato e dall'altro dell'interfaccia.
Il lavoro si è quindi concentrato inizialmente sugli aspetti geometrici del problema quindi sull'implementazione delle strutture dati necessarie alla gestione di una superficie bidimensionale definita da una funzione implicita immersa in una triangolazione 3D. L'implementazione che verrà presentata in questa tesi permette di gestire un numero imprecisato di interfacce attraverso il dominio con il vincolo che ciscun elemento non possa essere attraversato da più di due *level set*.
Si è poi passati all'implementazione di strutture dati che permettessero di gestire i gradi di libertà di uno spazio di approssimazione arricchito.
In seguito sono state implementate le routine necessarie per la costruzione del sistema lineare associato al problema discreto. Nell'ultima fase dell'implementazione, si è lavorato al fine di permettere la visualizzazione di una soluzione discontinua all'interno di un elemento.
Il codice è stato poi validato grazie ad alcuni casi test significativi e sono stati verificati i principali risultati teorici relativi alla convergenza e accuratezza del metodo.
Nel corso del lavoro è stata posta particolare attenzione ad un'implementazione che permettesse il calcolo parallelo pertanto è stata fatta anche un' analisi di scalabilità del codice.

L'ambiente in cui è stato svolto questo progetto è `LifeV` ([9], [10], [18]), una libreria di calcolo parallelo a Elementi Finiti scritta in linguaggio C++, al cui sviluppo collaborano il Politecnico di Milano, l'INRIA di Parigi, l'Emory University di Atlanta e l'EPFL di Losanna. Il progetto è iniziato nel 2002 e il codice è rilasciato sotto la licenza LGPL.
E' una libreria scritta in C++ [26] che fa uso di alcune librerie esterne, tra cui `Trilinos` [16] per il calcolo matriciale e `ParMetis` [17] per il partizionamento della matrice. `LifeV` è un codice parallelo che utilizza il protocollo MPI per la comunicazione attraverso i diversi processori.
Questa tesi si è basata sulla versione di sviluppo di `LifeV`, cosa che ha permesso di seguire le modifiche recenti apportate alla libreria, in particolar modo per quanto riguarda le novità riguardanti le strutture dati relative alla griglia di calcolo. La collaborazione con gli altri sviluppatori è stata resa possibile grazie all'utilizzo di `Git`, un programma concepito per la gestione delle versioni [25].
Esso permette infatti un'efficiente gestione dello sviluppo di un programma nel momento in cui a partire da una stessa base (*master*) gli sviluppatori portano avanti modifiche, aggiornamenti o novità a parti diverse del programma. Ogni gruppo di sviluppo genera un nuovo *branch*, ovvero una versione del programma originale su cui vengono operate le modifiche e le aggiunte, una volta che il *branch* è completo e stabile, esso può essere inglobato (*merge*) nel *master*. Inoltre chi sviluppa un *branch* può costantemente integrare le modifiche al *master* (o anche altri *branch*) assicurandosi così che il codice sia sempre il più possibile compatibile e coerente con lo sviluppo generale della libreria.
Quanto viene presentato in questa tesi è contenuto attualmente nel *branch XFEM* di `LifeV`, al quale si fa riferimento per quanto riguarda il codice implementato. Nei capitoli che seguono, si è cercato di minimizzare le linee di codice riportate, in quanto si è preferito spiegare le idee che stanno alla base di questa implementazione. Per tutte le strutture dati (classi) implementate in questo lavoro di tesi vengono dunque spiegate le funzioni dei principali metodi e attributi.

L'esposizione del lavoro è articolata come segue:

Nel primo capitolo viene presentato il problema modello che si intende risolvere e il metodo XFEM è descritto in dettaglio, riportando anche i principali risultati teorici di consistenza e convergenza.

Nel secondo capitolo viene descritta l'implementazione in `LifeV` della gestione di una *mesh* attraversata da un insieme di *level set*.

Nel terzo capitolo viene presentata l'implementazione degli algoritmi necessari all'integrazione del problema differenziale. Questo comprende quindi il partizionamento della *mesh*, la gestione dei gradi di libertà, l'integrazione sugli elementi attraversati dall'interfaccia e l'esportazione della soluzione.

Nel quarto capitolo sono mostrati e analizzati i risultati ottenuti circa la convergenza del metodo e le performance del codice implementato. Vengono evidenziati sia i punti in cui i risultati corrispondono alle attese, sia quelli in cui non sono pienamente soddisfacenti e si forniscono delle spiegazioni per quanto è stato ottenuto.

Nel capitolo conclusivo , dopo aver brevemente riassunto i risultati ottenuti in questa tesi, vengono date indicazioni circa i possibili sviluppi del lavoro svolto.

Capitolo 1

Il metodo degli Elementi Finiti Estesi

1.1 Un'introduzione al metodo

Il metodo degli Elementi Finiti Estesi (XFEM) permette di approssimare soluzioni poco regolari in alcune regioni del dominio grazie ad un arricchimento locale dello spazio di approssimazione rispetto alla base utilizzata nel metodo standard agli Elementi Finiti. In questi casi, il metodo standard agli elementi finiti (FEM) [20] è poco accurato mentre il metodo XFEM presenta significativi vantaggi sul piano dell'ordine di convergenza.
I campi di applicazione sono molteplici e spaziano dalla meccanica dei solidi (numerosi studi in questo campo sono stati fatti in anni recenti da Belytschko, [23]) a problemi di interazione fluido-struttura [12].
Alla base di questo metodo c'è il concetto di partizione dell'unità [24]. Una partizione dell'unità in un dominio Ω è un set di m funzioni μ_i tali che

$$\sum_{i=1}^{m} \mu_i(\mathbf{x}) = 1, \forall \mathbf{x} \in \Omega$$

Ogni funzione $\psi(\mathbf{x})$ può essere rappresentata attraverso l'applicazione della partizione dell'unità con ψ stessa dato che, ovviamente, vale:

$$\psi(\mathbf{x}) = \sum_{i=1}^{m} \mu_i(\mathbf{x})\psi(\mathbf{x}), \forall \mathbf{x} \in \Omega$$

Sia $\mathcal{I}$ l'insieme dei nodi in Ω e sia $\mathcal{I}_k^* \subset \mathcal{I}$ il sottoinsieme dei nodi sui quali si vuole localizzare l'arricchimento dato dalla funzione ψ^k. Una funzione ad elementi finiti U può essere dunque arricchita nel seguente modo:

$$U(\mathbf{x}) = \underbrace{\sum_{j \in \mathcal{I}} U_j \phi_j(\mathbf{x})}_{U^{FE}} + \underbrace{\sum_{k} \sum_{i \in \mathcal{I}_k^*} q_i^k \mu_i(\mathbf{x})[\psi^k(\mathbf{x}) - \psi^k(\mathbf{x}_i)]}_{U^{enr}} \tag{1.1}$$

dove ϕ_j sono le funzioni di base standard, U_j sono i valori nodali sui gradi di libertà standard, μ_i rappresenta una partizione dell'unità nel nodo i, ψ^k le funzioni di arricchimento e q_i^k dei parametri incogniti. La parte U^{FE} è quindi l'approssimazione ad elementi finiti standard

mentre U^{enr} è l'arricchimento ottenuto tramite la partizione dell'unità di una funzione di arricchimento ψ.
I valori nodali q_i^k sono parametri incogniti che aggiustano l'arricchimento in modo da approssimare al meglio la soluzione. Quando le funzioni μ_i hanno supporto piccolo e compatto [1] il sistema lineare ottenuto dalla discretizzazione sarà sparso. Una possibile scelta per la partizione dell'unità è porre $\mu_i = \chi_{\Omega_i}$ dove gli Ω_i sono m sottoinsiemi di Ω a due a due disgiunti e tali che $\bigcup_i^m \bar{\Omega}_i = \bar{\Omega}$ e χ_{Ω_i} è la funzione caratteristica di Ω_i.
La forma delle funzioni di arricchimento non deve necessariamente essere la stessa delle funzioni di base standard, sebbene nel metodo XFEM che verrà presentato si assumerà $\phi_j = \psi_k$.
Diverse tecniche di arricchimento sono state proposte e analizzate. Una panoramica abbastanza completa sullo stato attuale dei metodi XFEM può essere trovata in [11].
Nel corso di questo lavoro faremo riferimento a quanto presentato da Hansbo e Hansbo ([14],[13]), i quali hanno proposto un arricchimento basato sulla funzione di Heaviside negli elementi attraversati dall'interfaccia. Quando un elemento K viene attraversato da un'interfaccia definita dall'equazione $f(\mathbf{x}) = 0$, esso viene rimpiazzato da due elementi geometricamente identici, K' e K'', ciascuno con le proprie funzioni di base. Una funzione ad elementi finiti U in un elemento K intersecato dall'interfaccia è data dall'espressione:

$$U(\mathbf{x}) = \sum_{i \in K'} U_i \phi'_i(\mathbf{x}) H(f(\mathbf{x})) + \sum_{j \in K''} \tilde{U}_j \phi''_j(\mathbf{x})(1 - H(f(\mathbf{x}))) \tag{1.2}$$

dove quindi i cicla sui nodi dell'elemento K', j sui nodi dell'elemento K'' e $H(\cdot)$ è la funzione di Heaviside.
Questo corrisponde ad arricchire ogni elemento attraversato da una discontinuità usando le stesse funzioni di base del metodo standard. L'integrazione della forma debole su questi elementi richiede algoritmi speciali per via della presenza della funzione di Heaviside.
Difatti, negli elementi finiti standard, il calcolo della matrice di rigidezza dell'elemento e degli altri termini della forma debole richiede l'integrazione di funzioni polinomiali, calcolo che può essere svolto usando formule di quadratura standard, quali le formule di Gauss. Quando lo spazio di approssimazione è arricchito da funzioni discontinue (come la Heaviside), le usuali tecniche di quadratura devono essere modificate in modo da preservare l'accuratezza e l'ordine di convergenza del metodo. Diversi approcci sono proposti in letteratura tra cui:

- formule di quadratura di Gauss ad alto ordine,
- trasformazione in un integrale di linea o di superficie,
- integrazione del sottodominio, in cui l'elemento tagliato è ritriangolato in modo che la frontiera dei sotto-elementi sia allineata alla discontinuità e una formula di quadratura standard è poi usata su ciascun sotto-elemento.

L'implementazione che verrà presentata nei capitoli seguenti corrisponde a quest'ultima alternativa.

[1] Sono diverse da zero su un sottodominio di Ω.

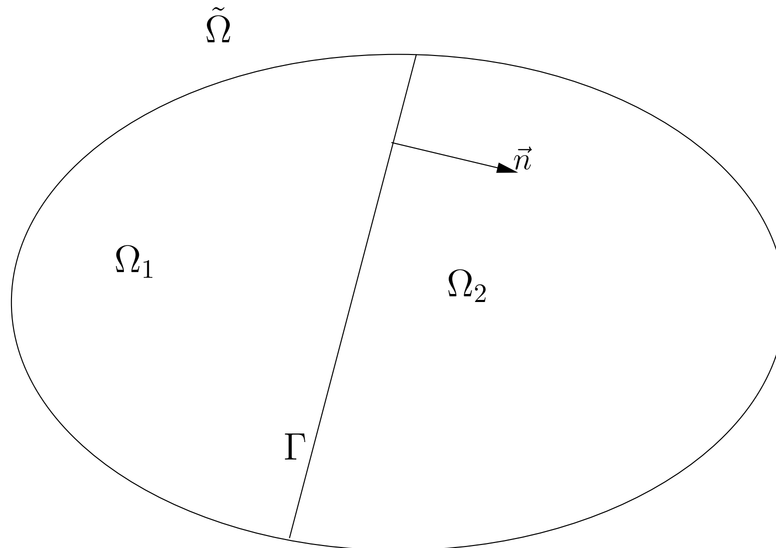


Figura 1.1: Dominio e superficie di discontinuità

1.2 Formulazione del problema

Consideriamo il seguente problema modello (si veda [14], [13]). Sia $\tilde{\Omega}$ un dominio limitato in $\mathbb{R}^3$, con frontiera poligonale convessa $\partial\tilde{\Omega}$ e Γ una frontiera interna regolare che divide $\tilde{\Omega}$ in due insiemi aperti Ω_1 e Ω_2 come in figura 1.1.

Per ogni funzione sufficientemente regolare u in $\Omega = \Omega_1 \cup \Omega_2$ definiamo il salto di u attraverso Γ come $[u] = u_1 \mid_\Gamma - u_2 \mid_\Gamma$, dove $u_i = u \mid_{\Omega_i}$ è la restrizione di u in Ω_i.
Consideriamo il seguente problema stazionario di trasmissione del calore con una discontinuità nella conduttività attraverso Γ e una condizione non omogenea sulla derivata conormale all'interfaccia:

$$\begin{aligned} & -\nabla \cdot (\alpha \nabla u) = f \text{ in } \Omega = \Omega_1 \cup \Omega_2 \\ & u = 0 \text{ su } \partial\tilde{\Omega} \\ & [u] = 0 \text{ su } \Gamma \\ & [\alpha \nabla_{\mathbf{n}}] = g \text{ su } \Gamma \end{aligned} \tag{1.3}$$

dove

$$\alpha = \begin{cases} \alpha_1 \text{ in } \Omega_1 \\ \alpha_2 \text{ in } \Omega_2 \end{cases}$$

Abbiamo indicato con $\mathbf{n}$ è la normale uscente Ω_1 e $\nabla_{\mathbf{n}} v = \mathbf{n} \cdot \nabla v$.
Nel problema modello considerato è stata imposta la continuità della soluzione attraverso Γ. La discontinuità è imposta invece sulla derivata conormale di u su Γ. In generale il metodo XFEM che verrà presentato è in grado di approssimare anche problemi con discontinuità nella soluzione, ma ai fini della validazione dell'implementazione è stato considerato un problema in cui la discontinuità è presente solo nella derivata.
Per il dominio connesso aperto limitato $\tilde{\Omega}$ usiamo gli spazi di Sobolev standard $H^r(\tilde{\Omega})$ con norma $\| \cdot \|_{r,\tilde{\Omega}}$ e gli spazi $H_0^r(\tilde{\Omega})$ con traccia nulla sulla frontiera $\partial\tilde{\Omega}$. Per l'insieme aperto $\Omega = \cup_{i=1}^2 \Omega_i$, dove Ω_i sono le due componenti disgiunte di Ω, sia $H^k(\Omega_1 \cup \Omega_2)$ lo spazio di

Sobolev di funzioni in Ω tali che $u|_{\Omega_i} \in H^k(\Omega_i)$ con norma:

$$\|\cdot\|_{k,\Omega_1\cup\Omega_2} = \left(\sum_{i=1}^{2} \|\cdot\|^2_{k,\Omega_i}\right)^{1/2}$$

Assumiamo $f \in L_2(\tilde{\Omega})$, $g \in H^{1/2}(\Gamma)$ e che α sia costante e positiva in ciascun Ω_i. Sotto le ipotesi di frontiera $\partial\tilde{\Omega}$ e interfaccia Γ sufficientemente regolari, il problema (1.3) ha una soluzione unica in H^2 su ciascun sottodominio e vale la seguente stima a priori ([22],[6]):

$$\|u\|_{1,\tilde{\Omega}} + \|u\|_{2,\Omega_1\cup\Omega_2} \leq C(\|f\|_{0,\tilde{\Omega}} + \|g\|_{1/2,\Gamma}) \tag{1.4}$$

dove C è una costante.
La formulazione debole del problema (1.3) è la seguente: trovare $u \in H^1_0(\tilde{\Omega})$ tale che

$$a(u,v)_{\tilde{\Omega}} = (f,v)_{\tilde{\Omega}} + (g,v)_\Gamma,\ \forall v \in H^1_0(\tilde{\Omega}) \tag{1.5}$$

dove $a(u,v) : H^1_0(\tilde{\Omega}) \times H^1_0(\tilde{\Omega}) \to \mathbb{R}$ è una forma bilineare definita nel modo seguente:

$$a(u,v)_{\tilde{\Omega}} = \int_{\tilde{\Omega}} \alpha \nabla u \nabla v d\mathbf{x}, \forall u,v \in H^1_0(\tilde{\Omega})$$

Vediamo ora come discretizzare il problema (1.5).

1.3 Il problema discreto

In un approccio standard agli Elementi Finiti, il salto nella derivata conormale quando $\alpha_1 \neq \alpha_2$ è preso in considerazione facendo coincidere Γ con le facce degli elementi della griglia e imponendo le condizioni di interfaccia sui nodi della griglia che giacciono su Γ. Questo può essere poco pratico quando si debbano, per esempio, eseguire calcoli con diverse configurazioni di Γ.
Quello che si vuole fare invece in un approccio XFEM è risolvere questo problema su una triangolazione conforme $\mathcal{T}_h$ di $\tilde{\Omega}$ che sia indipendente dalla posizione dell'interfaccia Γ. Vogliamo tuttavia permettere che la soluzione approssimata abbia derivata conormale discontinua negli elementi intersecati dall'interfaccia.
Per ogni elemento $K \in \mathcal{T}_h$, sia $K_i = K \cap \Omega_i$ (la parte di K contenuta in Ω_i). Definiamo con $\mathcal{G}_h = \{K \in \mathcal{T}_h : K \cap \Gamma \neq \emptyset\}$ l'insieme degli elementi che sono intersecati dall'interfaccia.
Siano le funzioni di base di un elemento $K \in \mathcal{G}_h$ lineari (considereremo di usare elementi finiti lineari) ma discontinue attraverso Γ. Quindi definiamo la generica funzione ad elementi finiti estesi su un elemento di $\mathcal{G}_h$:

$$\phi = \begin{cases} \phi_1 \text{ in } K_1 \\ \phi_2 \text{ in } K_2 \end{cases}$$

con ϕ_1 e ϕ_2 lineari.
Poiché ϕ è discontinua attraverso l'interfaccia non c'è a priori alcuna relazione tra i gradi di libertà di ϕ_1 e ϕ_2. Per determinare ϕ_1 sono necessari quattro gradi di libertà (in 3D, tre in 2D) e così pure ϕ_2. Pertanto in un elemento tagliato devono essere raddoppiati i gradi di libertà per poter definire ϕ.

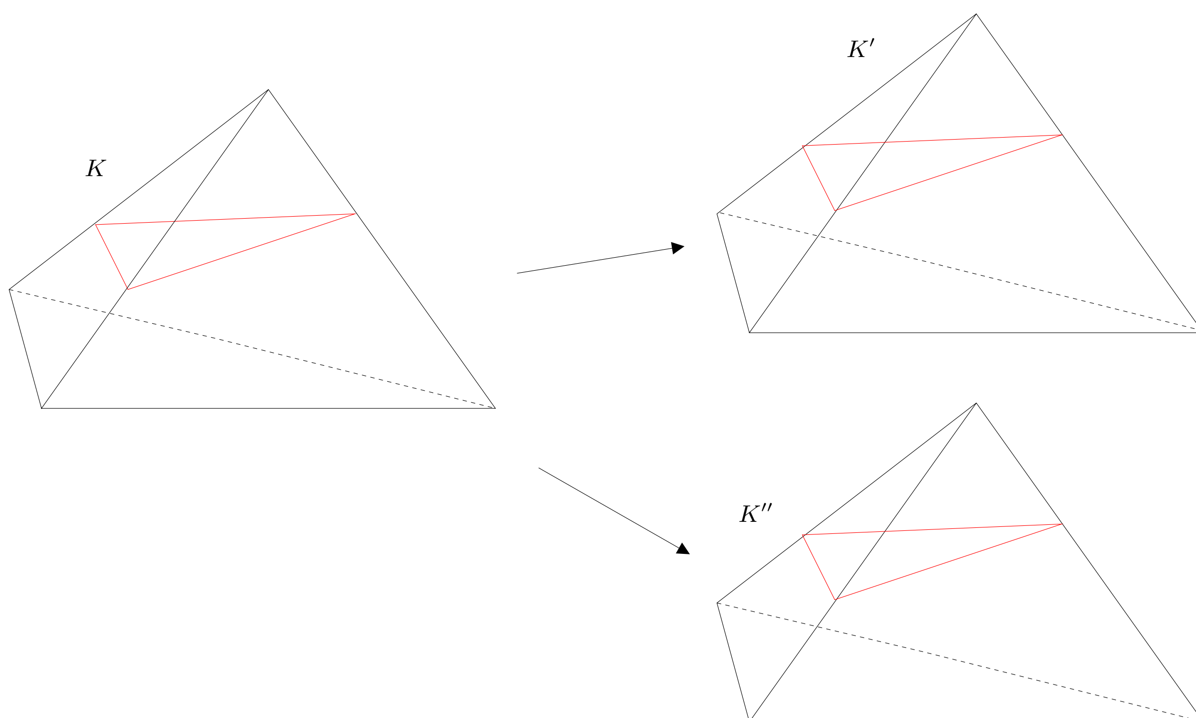


Figura 1.2: *Split* di un elemento attraversato da Γ

Un elemento K attraversato da Γ viene quindi sostituito da due elementi K' e K'', coincidenti geometricamente, ma ciascuno con i propri gradi di libertà nei nodi. Nella risoluzione del problema discreto i contributi relativi all'elemento K' saranno calcolati su quella parte di elemento ottenuta dall'intersezione di K con Ω_1 che abbiamo definito K_1. Discorso analogo vale per l'elemento K'', i cui contributi saranno calcolati su K_2.
Pertanto, a livello di notazione, con K' e K'' faremo riferimento agli elementi che hanno sostituito quello attraversato da Γ, ciascuno con i propri gradi di libertà. K_1 e K_2 sono invece le regioni $K_i = K \cap \Omega_i$ su cui verranno calcolati i contributi per la costruzione del sistema lineare associato al problema discreto (1.7).
La base dello spazio di approssimazione è dunque ottenuta a partire dalla base standard rimpiazzando per ogni nodo j di un elemento intersecato da Γ la sua funzione di base ϕ_j con due nuove funzioni di base ϕ'_j e ϕ''_j tali che:

$$\phi'_j = \phi\chi_{\Omega_1} \qquad \text{e} \qquad \phi''_j = \phi\chi_{\Omega_2}$$

dove χ_{Ω_i} è la funzione caratteristica di Ω_i.
Questa tecnica, che sdoppia l'elemento intersecato da Γ definendo nuove funzioni di base discontinue e raddoppiando i gradi di libertà permette di trascurare la geometria dell'interfaccia fino al momento dell'integrazione del problema differenziale.
Formalizzando quando detto sinora, cerchiamo una soluzione discreta $U = (U_1, U_2)$ nello spazio $V^h = V_1^h \times V_2^h$, dove:

$$V_i^h = \{w \in [H^1(\Omega_i)]^n : w\,|_{K\cap\Omega_i} \text{ è lineare}, \forall K \in \mathcal{T}_h \text{ e } \phi_i\,|_{\partial\tilde{\Omega}} = 0\} \tag{1.6}$$

per $i = 1, 2$.
In questo modo le funzioni in V^h potranno essere discontinue attraverso Γ.

Definiamo:

$$\kappa_i \mid_K = \frac{\mid K_i \mid}{\mid K \mid}$$

dove $\mid K \mid$ è la misura di K. Si avrà che $0 \le \kappa_i \le 1$ e $\kappa_1 + \kappa_2 = 1$. Definiamo inoltre questa combinazione convessa di $\phi \in V^h$ attraverso Γ:

$$\{\phi\} = (\kappa_1\phi_1 + \kappa_2\phi_2) \mid_\Gamma$$

Il metodo è definito dal problema variazionale seguente: trovare $U \in V^h$ tale che

$$a_h(U, \phi) = L(\phi),\ \forall \phi \in V^h \tag{1.7}$$

dove

$$a_h(U, \phi) = (\alpha_i \nabla U_i, \nabla \phi_i)_{\Omega_1 \cup \Omega_2} - ([U], \{\alpha \nabla_n \phi\})_\Gamma - (\{\alpha \nabla_n U\}, [\phi])_\Gamma + (\lambda [U], [\phi])_\Gamma, \tag{1.8}$$

$$L(\phi) = (f, \phi)_\Omega + (\kappa_2 g, \phi_1)_\Gamma + (\kappa_1 g, \phi_2)_\Gamma \tag{1.9}$$

e λ un parametro di penalità da fissare in modo opportuno (si veda 1.4).
Vale il seguente teorema [14]:

Teorema 1. *Il problema discreto* (1.7) *è fortemente consistente nel senso che, per u che risolve* (1.3)

$$a_h(u, \phi) = L(\phi), \forall \phi \in V^h$$

Come conseguenza vale l'ortogonalità di Galerkin, ovvero:

$$a_h(u - U, \phi) = 0, \forall \phi \in V^h$$

1.4 Analisi

Per ottenere delle stime sull'errore è necessario fare prima alcune ipotesi sull'intersezione tra Γ e gli elementi $K \in \mathcal{G}_h$.
Consideriamo una triangolazione conforme del dominio $\tilde{\Omega}$, basata su elementi tetraedrici.
Sia h_K il diametro di K e sia $h_{max} = \max_{K \in \mathcal{T}_h} h_K$. Sia inoltre ρ_K il diametro di K. Facciamo le seguenti ipotesi sulla *mesh* e l'interfaccia Γ:

A1 La triangolazione è regolare, dunque: $\exists C > 0$ tale che

$$h_K / \rho_K \le C,\ \forall K \in \mathcal{T}_h$$

A2 L'intersezione di Γ e la frontiera dell'elemento $K \in \mathcal{G}_h$ è una superficie o una faccia di K. Nel primo caso Γ interseca 3 o 4 lati del tetraedro, ciascuno in un punto.

A3 Per il caso in cui l'intersezione tra Γ e l'elemento è una curva interna ad esso, consideriamo il piano passante per 3 dei punti di intersezione con i lati, e sia $\Gamma_{K,h}$ l'intersezione tra K e questo piano. Allora Γ_K è una funzione su $\Gamma_{K,h}$ per un'opportuna scelta dei punti di intersezione; cioè

$$\Gamma_K = \{(\chi, \eta, \zeta) : (\chi, \eta, 0) \in \Gamma_{K,h}, \zeta = w(\chi, \eta)\} \tag{1.10}$$

in coordinate locali (χ, η, ζ).

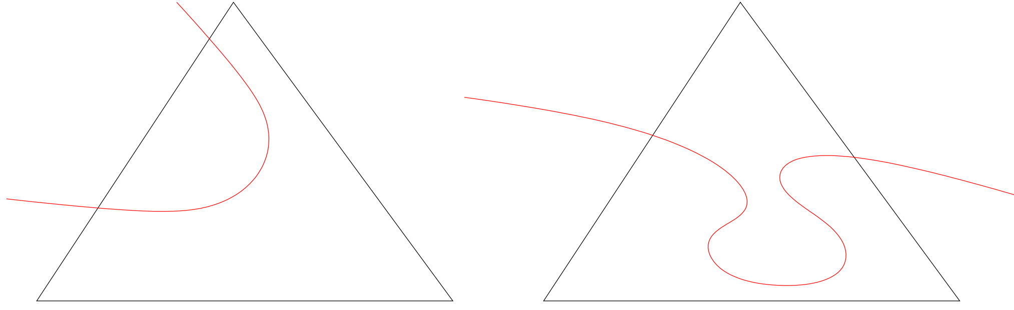


Figura 1.3: Elementi attraversati da Γ per cui non sono valide le assunzioni fatte

Le ultime due ipotesi sono soddisfatte per triangolazioni sufficientemente fini. L'idea è che la superficie che interseca l'elemento sia sufficientemente ben rappresentabile attraverso un piano. Si escludono quindi situazioni di una superficie che entra ed esce dall'elemento attraverso la stessa faccia.

Alcune delle possibili situazioni in cui queste ipotesi non sono verificate sono mostrate in figura 1.3 per il caso bidimensionale.

Fatte queste ipotesi possiamo ora enunciare alcuni importanti risultati teorici. Per le dimostrazioni si faccia riferimento a [14] e [13].

1.4.1 Proprietà di approssimazione dello spazio discreto V^h

Definiamo le seguenti norme sullo spazio V^h:

$$\|v\|_{1/2,h,\Gamma}^2 = \sum_{K\in\mathcal{G}_h} h_K^{-1}\|v\|_{0,\Gamma_K}^2 \tag{1.11}$$

$$\|v\|_{-1/2,h,\Gamma}^2 = \sum_{K\in\mathcal{G}_h} h_K\|v\|_{0,\Gamma_K}^2 \tag{1.12}$$

e

$$|||\ v\ |||^2 = \|\nabla v\|_{0,\Omega_1\cup\Omega_2}^2 + \|\{\nabla_{\mathbf{n}} v\}\|_{-1/2,h,\Gamma}^2 + \|[v]\|_{1/2,h,\Gamma}^2 \tag{1.13}$$

dove $\|v\|_{0,\Gamma_K} = \|\gamma_0 v\|_{L^2(\Gamma_K)}$ e $v\ |_{\Gamma_K} = \gamma_0 v$ è la traccia di una funzione $v \in V^h$ su Γ_K.
Notiamo che

$$(u,v)_\Gamma \leq \|u\|_{1/2,h,\Gamma}\|v\|_{-1/2,h,\Gamma}, \forall u,v \in V^h$$

Per mostrare che le funzioni in V^h approssimano le funzioni $v \in H_0^1(\tilde{\Omega}) \cap H^2(\Omega_1 \cup \Omega_2)$ all'ordine h nella norma $|||\cdot|||$, costruiamo un interpolante di v a partire dagli interpolatori nodali delle estensioni in H^2 di $v_i \in \Omega_i$ nel modo seguente. Supponiamo esista un operatore di estensione $\mathrm{E}_i : H^2(\Omega_i) \to H^2(\tilde{\Omega})$ tale che $(\mathrm{E}_i w)\ |_{\Omega_i} = w$ e

$$\|\mathrm{E}_i w\|_{s,\tilde{\Omega}} \leq C\|w\|_{\Omega_i}, \ \forall w \in H^s(\Omega_i), s = 0,1,2. \tag{1.14}$$

Sia $\mathcal{I}_h : C^0(\bar{\Omega}) \cap H^2(\tilde{\Omega}) \to V^h$ l'operatore standard di interpolazione nodale:

$$\mathcal{I}_h^* v = (\mathcal{I}_{h,1}^* v_1, \mathcal{I}_{h,2}^* v_2) \text{ dove } \mathcal{I}_{h,i}^* v_i = (\mathcal{I}_h \mathrm{E}_i v_i)\ |_{\Omega_i} \tag{1.15}$$

Vale il seguente teorema:

Teorema 2. *Sia* $\mathcal{I}_h^*$ *un operatore di interpolazione definito nella* (1.15). *Allora*

$$||| \, v - \mathcal{I}_h^* v \, ||| \leq C_1 h \|v\|_{2,\Omega_1 \cup \Omega_2}, \forall v \in H_0^1(\tilde{\Omega}) \cap H^2(\Omega_1 \cup \Omega_2) \tag{1.16}$$

dove la norma $||| \cdot |||$ *è definita in* (1.13).

1.4.2 Stima a priori

Innanzitutto mostriamo che la forma discreta (1.7) è continua e coerciva. Per fare questo facciamo uso del seguente lemma:

Lemma 1. *Vale la seguente disuguaglianza inversa:*

$$\|\{\nabla_{\mathbf{n}} \phi\}\|^2_{-1/2,h,\Gamma} \leq C_I \|\nabla \phi\|^2_{0,\Omega_1 \cup \Omega_2} \tag{1.17}$$

Lemma 2. *Dato* λ *sufficientemente grande, la forma discreta* $a_h(\cdot, \cdot)$ *è coerciva su* V_h*, cioè* $\exists C > 0$ *tale che*

$$a_h(v, v) \geq C \, ||| \, v \, |||^2, \forall v \in V^h, \tag{1.18}$$

La forma discreta è anche continua, cioè

$$a_h(u, v) \leq C \, ||| \, u \, |||||| \, v \, |||, \forall u \in V, \forall v \in V \tag{1.19}$$

Dalla dimostrazione, per cui rimandiamo a [14], si ricava una stima per λ:

$$\lambda = \frac{\gamma}{h_K} \tag{1.20}$$

dove $\gamma > 4C_I \max_\Omega \alpha$, per cui vale la coercività.

Teorema 3. *Sotto le assunzioni A1-A3, per* U *soluzione di* (1.7) *e* u *soluzione di* (1.3)*, valgono le seguente stime a priori:*

$$||| \, u - U \, ||| \leq Ch \|u\|_{2,\Omega_1 \cup \Omega_2} \tag{1.21}$$

e

$$\|u - U\|_{0,\Omega} \leq Ch^2 \|u\|_{2,\Omega_1 \cup \Omega_2} \tag{1.22}$$

1.4.3 Stima a posteriori

Consideriamo un funzionale dell'errore $\mathcal{J}(e)$ dove $e = U - u$ e definiamo gli stimatori locali e globali:

$$\begin{aligned} E_K(U) =& (h_K^2 \|f + \nabla \cdot (\alpha \nabla U)\|^2_{0,K_1 \cup K_2} + h_k \|[\alpha_i \nabla U_i]\|^2_{0,\partial K} \\ & - h_K \|g - [\alpha \nabla U]\|^2_{0,\Gamma_K} + h_K^{-1} \|[U]\|^2_{0,\Gamma_K})^{1/2} \end{aligned}$$

e

$$E(U) = \left(\sum_{K \in \mathcal{T}_h} h_K^2 E_K(U)^2 \right)^{1/2} \tag{1.23}$$

Allora vale la seguente stima a posteriori:

Teorema 4. *Per un funzionale lineare continuo* $\mathcal{J}(\cdot)$ *su* $L_2(\Omega)$*, sia* $J \in L_2(\Omega)$ *definito dal teorema di rappresentazione di Riesz, cioè* $\mathcal{J}(\cdot) = (J, \cdot)_\Omega$*. Esiste una costante positiva* C *tale che*

$$\mathcal{J}(e) \leq CE(U)\|J\|_{0,\Omega} \tag{1.24}$$

Questa stima può essere utilizzata per implementare algoritmi adattativi volti a minimizzare un dato funzionale $\mathcal{J}(\cdot)$ dell'errore ([20], [4]).

1.5 Cenni sull'implementazione

Ricordiamo che una base per V_h è ottenuta dalla base standard per gli Elementi Finiti sostituendo le funzioni di base standard su ogni elemento attraversato da Γ con coppie di funzioni di base ristrette ad Ω_1 e Ω_2, rispettivamente. Entrambe queste funzioni sono rappresentate dagli stessi nodi della triangolazione originale. I punti di intersezione tra i lati dell'elemento e la superficie non entrano in gioco nel definire l'espressione delle nuove funzioni di base e quindi la geometria dell'intersezione viene usata solo al momento di integrare i termini della forma bilineare.
Per implementare il metodo proposto, è necessario determinare innanzitutto l'insieme $\mathcal{G}_h$ degli elementi intersecati da Γ. L'interfaccia Γ è definita attraverso il metodo *level set*, per cui la superficie attraverso la quale imporremo il salto della derivata conormale corrisponde alla curva di livello nulla di una funzione $f : \tilde{\Omega} \to \mathbb{R}$.
Ogni $K \in \mathcal{G}_h$ viene suddiviso in due copie identiche K' e K'', assegnate rispettivamente a quella parte di K che si trova in Ω_1 e Ω_2.
Per valutare numericamente $a_h(\cdot,\cdot)$ e $L(\cdot)$ operiamo nel seguente modo. Gli elementi che non sono attraversati da Γ vengono integrati nel modo usuale. Invece, in un elemento $K \in \mathcal{G}_h$ e *splittato* in K' e K'' si integra separatamente sulla regione K_1 dell'elemento K' e sulla regione K_2 dell'elemento K''.
L'integrazione sulle due parti dell'elemento è fatta con le consuete formule di quadratura gaussiane sugli elementi che costituiscono la ritriangolazione di Delaunay di K_1 e K_2. Per i dettagli rimandiamo ad capitolo 3.
Per la valutazione dei termini di salto in cui compare l'integrale su Γ_K si usano le formule di quadratura di Gauss per le entità di bordo (facce dell'elemento nel caso 3D) applicate a $\bar{K}_1 \cap \bar{K}_2$.
I contributi calcolati vengono poi assemblati coerentemente con l'attribuzione dei gradi di libertà, cosa resa complessa dal fatto che lo *splitting* degli elementi tagliati genera due elementi diversi per quanto riguarda i gradi di libertà ma coincidenti geometricamente.

Tagli multipli Se sono presenti più *level set*, la filosofia del metodo è la medesima: i gradi di libertà devono essere aumentati in modo opportuno. Per i dettagli sulla gestione di tagli multipli all'interno di un elemento si veda [8].
L'implementazione che viene proposta nei capitoli successivi è in grado di gestire n interfacce attraverso il dominio con il vincolo che uno stesso elemento non sia attraversato da più di due *level set*. La gestione dei gradi di libertà e gli algoritmi di integrazione tengono infatti conto della possibilità di avere anche due *level set* attraverso l'elemento.

Capitolo 2

Gestione di superfici in griglie 3D

2.1 Discontinuità attraverso il dominio

Una discontinuità può essere definita come un rapido cambiamento di una quantità attraverso una regione le cui dimensioni sono trascurabili rispetto a quelle del dominio, ad esempio un piano in un dominio tridimensionale.
La descrizione di una discontinuità nel contesto degli XFEM, sia essa nella soluzione u o in una sua derivata, è spesso realizzata attraverso il metodo *level set* [19]. Una funzione *level set* è una funzione scalare $f(\mathbf{x})$ la cui curva di livello nulla $f(\mathbf{x}) = 0$ è interpretata come la superficie di discontinuità. Come conseguenza, il dominio è diviso in due sottodomini Ω_1 e Ω_2, dove la funzione *level set* è rispettivamente positiva o negativa.

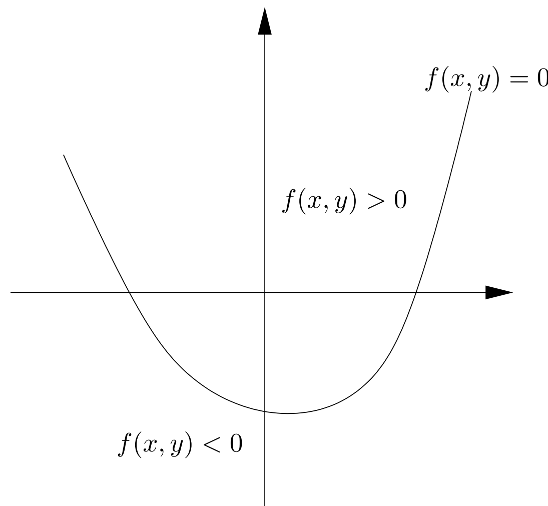


Figura 2.1: *Level set* in 2D

Ad esempio, una superficie di discontinuità sferica di raggio r attorno all'origine può essere definita dalla funzione *level set*:

$$f(x, y, z) = x^2 + y^2 + z^2 - r^2$$

La superficie di discontinuità sarà data da $\Gamma = \{\mathbf{x} \in \tilde{\Omega} \text{ tali che } f(\mathbf{x}) = 0\}$.
Questo metodo permette di determinare facilmente in quale semispazio si trova un punto $P_0 =$

$(x_0, y_0, z_0)^T$ semplicemente osservando il segno di $f(x_0, y_0, z_0)$. Questa semplice proprietà è ampiamente utilizzata nell'implementazione che viene ora presentata.

Per meglio comprendere ciò che è stato implementato al fine di gestire la presenza di *level set* che intersecano una griglia 3D in `LifeV` è prima necessario fornire una panoramica generale sulle strutture già presenti nella libreria.
In `LifeV` la geometria di un problema, quindi la *mesh* su cui viene risolta l'equazione differenziale è implementata attraverso le strutture dati seguenti:

- classi che definiscono la *mesh* i suoi elementi geometrici: **RegionMesh3D**, **MeshElementMarkedXD** e tutte le classi da cui queste ereditano. Ogni entità geometrica, dal punto sino alla *mesh* intera, ereditando dalla classe **MeshEntity**, possiede un ID globale, un ID locale e una *flag*. Tale *flag* contiene informazioni su alcune caratteristiche dell'entità geometrica, come ad esempio l'appartenenza alla frontiera del dominio o alla frontiera di una partizione. Il valore CUTTED di questa *flag* è stata introdotto per indicare in un calcolo di tipo XFEM un'entità geometrica attraversata da una discontinuità[1].
  La differenza tra ID locale e globale entra in gioco nel momento in cui si svolge un calcolo in parallelo, nel quale la *mesh* viene partizionata: l'ID locale verrà difatti assegnato dal processore su cui si trova l'entità geometrica (per svolgere i calcoli sulla singola partizione), mentre l'ID globale servirà poi per l'assemblaggio della soluzione.

- classe **Marker**, da essa ereditano tutte le classi che definiscono le entità geometriche in una *mesh*. Tale classe giocherà un ruolo chiave nell'implementazione degli XFEM in `LifeV`; il suo contenuto e funzionamento verrà trattato in dettaglio più avanti, nella sezione 2.3.1.

- classi che gestiscono la *mesh* come, ad esempio, **MeshUtility**.

Sono inoltre presenti alcuni *namespace* contenenti degli strumenti utili per la gestione delle strutture dati relative alle entità geometriche come, ad esempio, la routine **checkMesh3D()** che permette di testare la coerenza di una *mesh* e, se possibile, di correggere eventuali errori.
Su queste strutture dati e su questi strumenti si è basata la prima parte del lavoro di implementazione.
Nel seguito del capitolo verranno presentate in dettaglio tutte le strutture dati, implementate nel corso di questa tesi, che permettono di ottenere le informazioni geometriche necessarie per un calcolo di tipo XFEM come è stato presentato nel capitolo precedente.
L'idea generale è stata quella di implementare da un lato delle classi che gestiscono le superfici e dall'altro una classe contenente gli algoritmi che permettono di fare i calcoli necessari per determinare come la griglia viene tagliata inserendo poi le informazioni ottenute in opportune strutture dati, che, come vedremo, sono classi figlie di **Marker**. In questo modo tutta la parte algoritmica è esterna alla classe **RegionMesh3D** ma le informazioni necessarie alla risoluzione del problema differenziale sono contenute in essa.
Il codice scritto è così strutturato:

- classi che definiscono superfici data l'espressione della funzione *level set* $f(\mathbf{x})$ e che forniscono metodi per determinare posizioni dei punti e intersezioni;

[1]Questa *flag* è stata aggiunta nel corso del lavoro di tesi nell'ambito della revisione di alcune classi che gestiscono le entità della *mesh*.

- classe **Surfaceset**, contenitore di superfici;
- nuovi *marker* con informazioni aggiuntive per lati, facce ed elementi;
- classe **XfemMeshHandler** contenente le routine necessarie a tagliare la *mesh* inserendo le informazioni necessarie per il calcolo ad Elementi Finiti Estesi nelle strutture dati opportune (classi figlie di **Marker**).

Analizziamo ora nel dettaglio le strutture dati e gli algoritmi implementati in questa tesi.

2.2 Superfici bidimensionali e funzioni *level set*

2.2.1 La classe AnalyticSurface

E' un classe astratta pertanto non è possibile istanziare un oggetto di questa classe. Contiene gli attributi generici che servono a definire una superficie:

- **M_id**, un intero contenente l'ID della superficie;
- **M_coefficients**, vettore di *double* contenente i coefficienti dell'espressione analitica della superficie;
- due costruttori, uno di copia e uno dato il vettore di coefficienti. Dato quest'ultimo costruttore, al momento dell'istanziazione di un oggetto figlio di **AnalyticSurface**, vengono definiti i coefficienti presenti nell'espressione $f(\mathbf{x})$ nel caso in cui essa sià polinomiale;
- metodi che modificano l'ID e il vettore di coefficienti;
- **operator()**, metodo virtuale a cui è stato assegnato il valore nullo, deve essere definito nelle classi figlie di **AnalyticSurface**. Dovrà restituire, dato un punto $\mathbf{x}$, la valutazione $f(\mathbf{x})$ dove $f(\mathbf{x}) = 0$ è l'espressione analitica che definisce la superficie;
- **relativePosition()**, metodo che chiama **operator()** e restituisce [2] 0 se $\mid f(\mathbf{x}) \mid < toll$, -1 se $f(\mathbf{x}) < -toll$ e 1 se $f(\mathbf{x}) > toll$ dove la tolleranza utilizzata è contenuta nella variabile **S_distanceTolerance**;
- **intersection()**, metodo che, dati due punti in ingresso[3], cerca l'intersezione tra il segmento congiungente tali punti (in genere un lato della griglia) e la superficie utilizzando il metodo di bisezione. Nelle classi figlie, qualora si conoscano espressioni analitiche del punto di intersezione tra segmento e superficie, è possibile utilizzare l' *overridding* per velocizzare il calcolo evitando l'uso del metodo di bisezione;
- **S_distanceTolerance**, **S_bisectionTolerance** e **S_maxIter**, variabili statiche (quindi comuni a tutti gli oggetti della classe **AnalyticSurface**). La prima serve per valutare quando un punto è da considerarsi appartenente alla superficie, le altre due sono utilizzate nel metodo di bisezione.

[2] I valori di ritorno che vengono presentati sono raggruppati nell'*enum* **position_Type**, in modo da rendere più chiaro all'utente il contenuto della variabile.

[3] Oggetti della classe **MeshVertex** o figli.

A questi vanno aggiunti i metodi protetti **midPoint()** e **distance()** utilizzati nel metodo di bisezione e un metodo **showMe()** che mostra all'utente il contenuto delle variabili.

Alcuni esempi di classi derivate: Plane e Sphere Sono state implementate due classi figlie di **AnalyticSurface**, **Plane** e **Sphere**, attraverso il meccanismo della ereditarietà pubblica.
Nella classe **Plane** l'**operatore ()** restituirà:

$$f(x, y, z) = ax + by + cz + d$$

mentre nella classe **Sphere** restituirà:

$$f(x, y, z) = x^2 + y^2 + z^2 + ax + by + cz + d$$

Nella classe Plane, inoltre, il metodo **intersection()** è stato riscritto (*overidding*) sfruttando il fatto che è nota l'espressione analitica del punto di intersezione. Dati infatti $P_0 = (x_0, y_0, z_0)^T$ e $P_1 = (x_1, y_1, z_1)^T$ si cerca s con $0 < s < 1$ tale che:

$$a(x_0 + s(x_1 - x_0)) + b(y_0 + s(y_1 - y_0)) + c(z_0 + s(z_1 - z_0)) + d = 0$$

da cui si ricava:

$$s = \frac{-d - ax_0 - by_0 - cz_0}{a(x_1 - x_0) + b(y_1 - y_0) + c(z_1 - z_0)}$$

Dato s si calcola il punto di intersezione (valore di ritorno del metodo).
Per aggiungere nuovi tipi di superficie è quindi necessario definire una nuova classe **NewSurface** fornendo i costruttori opportuni e l'**operator()**:

```
class NewSurface: public AnalyticSurface{

typedef AnalyticSurface AnalyticSurface_Type;
typedef AnalyticSurface_Type::point_Type point_Type;
typedef AnalyticSurface_Type::vectorCoeff_Type vectorCoeff_Type;

NewSurface( const vectorCoeff_Type& coefficients );
NewSurface( const NewSurface& surface);

virtual Real operator()( const point_Type& point ) const;
};
```

Eventualmente si può ridefinire anche il metodo **intersection()**.

2.2.2 La classe SurfaceSet

Questa classe vuole essere un contenitore di superfici (che nel nostro caso sono oggetti di classi figlie di **AnalyticSurface**). Per sfruttare il polimorfismo, **SurfaceSet** contiene al suo interno un vettore di *shared pointer* a **AnalyticSurface** e metodi per gestire questo vettore (aggiunta di superfici, **getSurface()** che ritorna la superficie dato l'ID, etc.). Da notare che le superfici vengono aggiunte in modo che l'ID corrisponda alla posizione nel vettore in modo facilitarne l'estrazione.
Vi è inoltre un metodo **showMe()** che richiama il metodo **showMe()** di tutte superfici contenute nel vettore.

2.3 Informazioni sul taglio degli elementi

2.3.1 La classe Marker e classi figlie

Per capire come estendere le funzionalità della classe **RegionMesh3D** e dei suoi elementi è necessario vedere attentamente il ruolo dei *marker*.
La classe **Marker** di base è una classe contenente al suo interno un solo attributo protetto, **M_markerID** e i metodi per modificare tale attributo (nel rispetto di una certa *policy* passata attraverso il *parametro template* **FlagPolicy**). Nel *trait* **MarkerCommon** sono definiti i *marker* di base per le diverse entità geometriche, che non sono altro che ridefinizioni (*typedef*) della classe **Marker**.
Un particolare **MarkerCommon** è poi definito come **defaultMarkerCommon_Type** ed è quello usato di default dalle entità geometriche di `LifeV`.
RegionMesh3D e le altre entità geometriche (ad eccezione dell'entità 0D) sono classi templetizzate rispetto a due parametri: *GeoShape* attraverso il quale si definisce la geometria delle entità che costituiscono la mesh e *MC* attraverso il quale si definisce il **MarkerCommon** (in generale si usa **defaultMarkerCommon_Type**). L'entità 0D è templetizzata solamente rispetto a *MC*.
Tutte le entità geometriche ereditano poi pubblicamente dal corrispondente *marker* definito dal parametro *MC*. In questo modo tutte le entità geometriche della *mesh*, essa stessa compresa, ereditano dalla classe **Marker**. Abbiamo così che **MeshElementMarked0D** deriva dalla classe **MeshVertex** (parte geometrica dell'entità e attributi generali contenuti in **MeshEntity**) e **Marker** (contenente una flag), **MeshElementMarked1D** deriva da **MeshElement** e **Marker**, e via dicendo.
Creando delle nuove classi *marker* che ereditano dalla classe **Marker** in modo da mantenere immutate le funzionalità di base, e definendo un nuovo **MarkerCommon** è possibile estendere le informazioni contenute nella *mesh* e in tutte le entità geometriche da essa gestite. Vediamo un esempio di codice per un nuovo *marker* per gli elementi. Il nuovo *marker* sarebbe così definito:

```
template <typename FlagPolicy=EntityFlagStandardPolicy>
class NewMarkerElement<FlagPolicy>:public Marker<FlagPolicy>
{
public:
NewMarkerElement();
// altri costruttori e metodi...

private:
var_Type var1;
// altre variabili e metodi privati...
};
```

E la definizione del nuovo **MarkerCommon** sarebbe:

```
template
<class MT>
class NewMarkerCommon
```

```
{
public:

    typedef MT markerTraits_Type;
    typedef Marker<MT>  pointMarker_Type;
    typedef Marker<MT>  edgeMarker_Type;
    typedef Marker<MT>  faceMarker_Type;
    typedef NewMarkerElement<MT>  volumeMarker_Type;
    typedef Marker<MT>  regionMarker_Type;
};

typedef NewMarkerCommon<EntityFlagStandardPolicy> NewMarkerCommon_Type;
```

In questo progetto è stato necessario creare un *marker* arricchito per lati, facce ed elementi. Sono state quindi definite le classi **MarkerEdgeXfem**, **MarkerFaceXfem** e **MarkerElementXfem**. Queste classi derivano dalla classe **MarkerXfem** che a sua volta eredita da **Marker**[4].

E' stato poi definito un **xfemMarkerCommon_Type** come *typedef* del *trait* **MarkerCommonXfem** (si vedano i file MarkerXfem.hpp e MarkerDefinitions.hpp).
Tutti i nuovi *marker* sono stati forniti di due costruttori: quello vuoto e quello di copia, che richiamano i rispettivi costruttori del *marker* di base. Non sono stati forniti altri costruttori dal momento che all'atto della creazione della *mesh* non sono note le informazioni sui tagli, pertanto all'istanziazione di un **MeshElementMarkedXD** è necessario inizializzare gli attributi aggiuntivi a valori di default. Il costruttore di copia è invece necessario per garantire che le informazioni sui tagli non vengano perse durante il partizionamento.

La classe MarkerEdgeXfem Rispetto al marker di base contiene in più una multimappa della *standard library.* In ogni coppia della multimappa **M_pointsOnEdge**, il primo *unsigned int* è l'ID della superficie che taglia il lato e il secondo *unsigned int* è l'ID del punto di intersezione.
In questo modo per ogni lato sappiamo quante sono le superfici che lo intersecano e, dato l'ID, possiamo ricavare le informazioni circa il punto di intersezione (contenuto nel *marker* dell'elemento, come si vedrà in seguito). Questa variabile permette dunque di calcolare una sola volta il punto di intersezione tra l'*edge* e una data superficie. Difatti, al momento di cercare l'intersezione, verrà prima controllato se è già presente nella mappa l'ID della superficie intersecante, in tal caso viene utilizzato il punto già calcolato di cui è noto l'ID.
Per fare questo nella classe sono stati implementati i metodi necessari alla modifica e lettura della variabile **M_pointsOneEdge**.

La classe MarkerFaceXfem Il *marker* per le facce contiene la variabile **M_pointOnFace** in cui viene inserito l'ID di un eventuale punto di intersezione tra due superfici intersecanti e

[4]La classe **MarkerXfem** non contiene informazioni circa la geometria delle intersezioni ma il suo contenuto è necessario per una corretta assegnazione dei gradi di libertà, per tale motivo non viene presentata in dettaglio in questo capitolo.

la faccia stessa [5].
MarkerFaceXfem contiene inoltre la multimappa **M_subFaces** costituita da *pair* di *unsigned int* e **BareFace**. L'intero rappresenta la regione (vedi il paragrafo 2.4.4) in cui si trova la **BareFace**, che è una *struct* definita in MeshElementBare.hpp contenente solamente gli ID dei suoi vertici. Se la faccia non è tagliata da alcuna superficie tale mappa rimane vuota. Qualora invece la faccia sia tagliata in due o più parti (caso di taglio doppio), ciascuna parte viene ritriangolata e le queste *sub-face* sono inserite nella mappa aggiungendo l'informazione relativa alla regione. Le informazioni contenute in questa variabile sono importanti nel caso dell'iperbolico in cui un pezzo di elemento tagliato comunica con l'elemento adiacente tramite il solo pezzo di faccia comune ai due elementi e appartenente alla medesima regione.
Sono stati inoltre implementati tutti i metodi necessari alla modifica e lettura delle variabili contenute in questa classe tra cui metodi per aggiungere e togliere *sub face* (si veda il file MarkerXfem.hpp).

La classe MarkerElementXfem E' il *marker* associato all'entità d dimensionale. In questa classe troviamo le seguenti variabili:

- **M_weight**, variabile intera che contiene il peso dell'elemento, corrisponde al numero di parti in cui l'elemento è diviso dalle superfici che lo intersecano (si veda la sezione 2.4)
- **M_region**, *flag* che associa all'elemento la regione in cui si trova(si veda la sezione 2.4.4)
- **M_subElements**, multimappa della *standard library* contenente i sotto elementi (oggetti della classe **XfemCutElmentData**, si veda la sezione 2.3.2) associati alla regione di appartenenza (*unsigned int*, primo valore di ciascun *pair* della mappa)
- **M_cuttingFaces**, multimappa contenente le superfici di taglio, cioè le triangolazioni delle parti di piano che approssimano le superfici intersecanti l'elemento (vedi 2.4.3), associate alla regione di appartenenza
- **S_intersectionPoints**, vettore dei punti di intersezione tra la *mesh* e le superfici intersecanti[6].

All'interno della classe sono poi forniti i metodi per modificare queste variabili.
Si è scelto di utilizzare una variabile statica per i punti di intersezione (**S_intersectionPoints**) in modo che essa sia comune a tutti gli elementi e non sia replicata in ciascuno di essi (con conseguente spreco di memoria).
La scelta di archiviare le informazioni sui sotto-elementi in una multimappa è dovuta al fatto che nel corso dell'integrazione del problema differenziale sarà richiesto di avere i sotto-elementi associati ad una regione in modo da integrare su tutta una parte di elemento. Questo è stato implementato nel metodo **getRegionSubVolumes()** [7] che data in input la regione restituisce

[5] Dal momento che non sono previsti casi in cui tre superfici diverse attraversano la faccia, si avrà al massimo un punto di intersezione di questo tipo su una data faccia.

[6] I punti sono oggetti della classe **MeshVertex**, quindi per essi sono definiti gli ID, le coordinate x, y, z e la variabile **M_flag** (ereditata da **MeshEntity**) ma, rispetto alla struttura dati **MeshElementMarker0D** che rappresenta un punto in una *mesh* manca il *marker*.

[7] E' stato implementato anche il metodo analogo **getRegionCuttingFaces()** per le approssimazioni piane delle superfici che tagliano un elemento.

i *sub-element* corrispondenti.
Il calcolo del peso dell'elemento, quindi l'assegnazione della variabile **M_weight**, è fatta contando il numero di *keys* presenti nella multimappa **M_subElements**, quindi il numero di regioni diverse presenti in un elemento.
Oltre a quelli già indicati sono stati implementati i metodi necessari alla lettura e modifica di tutte le variabili della classe tra cui nominiamo **addSubVolume()** per l'aggiunta di *sub element* alla variabile **M_subElements**, **addIntersectionPoint()** per l'aggiunta di un punto di intersezione alla variabile statica della classe e **addCuttingFace()** per l'aggiunta delle facce 'di taglio' (sezione 2.4.3).

2.3.2 La classe XfemCutElementData

Gli elementi tagliati, come già visto, sono ritriangolati in modo da poter usare le consuete formule di quadratura sui sotto-elementi. E'stato quindi necessario definire una classe contenente gli ID globali e le coordinate dei vertici del sotto elemento (siano essi punti della *mesh* originale o punti di intersezione contenuti nel *marker* dell'elemento), e un ID che sarà utilizzato al momento dell'esportazione della soluzione sulla *mesh* tagliata.
Sono forniti tre costruttori (vuoto, di copia e dato ID e vettore di punti) e i metodi per modificare gli attributi protetti.

2.4 Algoritmi di taglio: la classe XfemMeshHandler

E'la classe che contiene tutti gli algoritmi che servono a determinare come la griglia viene tagliata dalle superfici e che si occupa di creare una *mesh* adatta all'esportazione della soluzione di un problema differenziale risolto con gli XFEM.
E' templetizzata rispetto a *GeoShape* ovvero rispetto alla geometria degli elementi che formano la *mesh*. E' in realtà un falso *template*, in quanto, allo stato attuale, gli algoritmi implementati si limitano a gestire elementi tetraedrici. La scelta di passare come parametro *template* la geometria delle entità che compongono la *mesh* permette di poter estendere più facilmente le funzionalità di **XfemMeshHandler** ad altri tipi di geometrie.

2.4.1 Il contenuto della classe

Le variabili All'interno di questa classe troviamo le seguenti variabili:

- **M_mesh**, è uno *shared pointer* ad un oggetto di classe **RegionMesh3D** [8];
- **M_surfaces**, è uno *shared pointer* ad un oggetto di classe **SurfaceSet**;
- **M_partMesh**, è uno *shared pointer* ad un oggetto di classe **MeshPartitioner**;

[8] Ricordiamo che **RegionMesh3D** è una classe templetizzata rispetto a due parametri: *GeoShape* e *MC*. **XfemMeshHandler** opera con *mesh* specializzate rispetto al secondo parametro template, dove *MC* è l' **xfemMarkerCommon_Type**, mentre il primo parametro è lo stesso rispetto a cui è templetizzata la classe **XfemMeshHandler**. La specializzazione rispetto a *MC* garantisce il fatto di avere una *mesh* le cui strutture dati sono arricchite per mezzo dei nuovi *marker*, come è stato indicato nella sezione 2.3.1.

- le mappe della *standard library* **M_newPointsOnEdges** e **M_newPointsOnFaces**, attraverso le quali, dato l'ID di un punto di intersezione è possibile ottenere l'ID del lato o della faccia a cui esso appartiene [9];
- **M_diffPointsTolerance**, un *double* contenente la distanza minima tra due punti perché essi possano essere considerati distinti, in genere viene inizializzata con il valore contenuto in **S_bisectionTolerance** della classe **AnalyticSurface** se si utilizza in costruttore non vuoto di **XfemMeshHandler** (di default pari a 10^{-5}), ma può essere in ogni caso modificata dall'utente;
- **M_volumeTolerance**, un *double* contenente la tolleranza al di sotto della quali un *sub-element* viene considerato a volume nullo. E' inizializzata con il cubo della tolleranza **M_diffPointsTolerance**;
- **M_areaTolerance** un *double* contenente la tolleranza al di sotto della quali una *sub-face* viene considerata ad area nulla. E' inizializzata con il quadrato della tolleranza **M_diffPointsTolerance**;
- **M_mapCutEle**, mappa contenente gli ID degli elementi tagliati e gli ID delle superfici intersecanti;
- **M_alreadyCutFaces**, *set* della *standard library* in cui vengono inseriti gli ID delle facce già ritriangolate dopo il taglio, in modo da non ripetere i calcoli due volte per ogni faccia interna;
- i contatori **M_notEntirelyCutElements**, **M_nullVolumes**, **M_elementsCounter**. Il primo conta gli elementi che vengono tagliati da una superficie su una faccia (non vengono dunque attraversati) [10], il secondo conteggia i volumi nulli creati dalla triangolazione di `QHull` (vedi 2.4.2) e il terzo serve per attribuite in modo incrementale l'ID globale dei sotto elementi;
- **M_cuttingFacesArea**, una mappa contenente il valore totale delle aree delle facce di taglio assegnate ad una certa regione (si veda 2.4.3).

Si tratta in generale di variabili funzionali al calcolo delle intersezioni, dei *sub-element* e delle *sub-face*; tutte le informazioni geometriche che devono essere conservate per la soluzione numerica del problema differenziale vengono invece inserite nei *marker* già presentati.

I costruttori Sono stati forniti due costruttori alla classe **XfemMeshHandler**. Un costruttore dati gli *shared pointer* alla *mesh* e al **SurfaceSet**, e un costruttore vuoto.
Il primo inzializza le variabili **M_mesh** e **M_surfaces** con i parametri dati in input e attribuisce i valori delle tolleranze a partire da quelle contenute negli oggetti di classe **AnalyticSurface**.

[9] Queste mappe sono utili per assegnare un valore alla variabile **M_flag** contenuta nel *marker* di base del nuovo punto

[10] Si generano quindi due elementi adiacenti tra i quali passa la superficie di discontinuità, la logica Xfem prevederebbe che entrambi gli elementi vengano raddoppiati ma in entrambi si genererebbe un elemento nullo. Vengono conteggiati tali casi degeneri, i quali nell'implementazione della parte differenziale non verranno presi in considerazione, arrestando l'esecuzione del programma.

Automaticamente viene modificata la variabile **M_elementsCounter**, e gli altri contatori vengono inizializzati a zero.
Se si usa il costruttore vuoto è necessario chiamare i metodi **setMesh()**, **setSurfaceSet()** e **setElementsCounter()**.
Le tolleranze possono essere in ogni caso modificate rispetto ai valori di default utilizzando i metodi **setVolumeTolerance()**, **setAreaTolerance()** e **setDiffPointsTolerance()**.

Il metodo getPoint() Questo metodo, dato l'ID, restituisce il punto corrispondente, cercandolo all'interno della *mesh* originale se l'ID è inferiore al numero di punti memorizzati nell'oggetto di classe **RegionMesh3D** altrimenti lo cerca nella variabile **S_intersectionPoints** nel *marker* dell'elemento. E' un metodo fondamentale per gestire i punti avendo due diversi contenitori: uno per i punti della *mesh* e uno per i nuovi punti di intersezione che vengono calcolati.

Metodi per il taglio degli elementi Per la presentazione e trattazione dei metodi necessari al calcolo delle intersezioni e dei *sub-element* rimandiamo alle sezione successive.

2.4.2 Il taglio degli elementi

Linea generale Il taglio della *mesh* è fatto chiamando la routine **intersectMeshSurfaces()**. Tale routine cicla su tutti gli elementi chiamando prima il metodo **isElementCut()** e poi **cutElement()**.
Il metodo **isElementCut()** individua quali sono gli elementi tagliati mentre **cutElement()** si occupa di calcolare le intersezioni e suddividere l'elemento tagliato negli opportuni *sub-element* a partire dal numero di superfici intersecanti (si veda 2.4.2).

Individuazione degli elementi tagliati Consideriamo l'elemento tagliato quando l'intersezione tra i lati e la superficie è costituita da almeno 3 punti. Possiamo distinguere i seguenti casi:

- elemento tagliato internamente
- taglio non interno, cioè superficie passante per una faccia dell'elemento (i tre vertici di quest'ultima si trovano sulla superficie). Il numero di questi elementi viene conteggiato nella variabile **M_notEntirelyCutElements**.

In figura 2.2 viene mostrato come gli elementi possono essere tagliati. I punti segnati in rosso sono quelli dati dall'intersezione con il *level set*. La terza immagine mostra cosa si intende per taglio non interno mentre le altre due mostrano i due casi più generali di intersezione tra *level set* ed elemento.
Per determinare se l'elemento è tagliato da una data superficie si cicla sui vertici e si chiama il metodo **relativePosition()** della classe **AnalyticSurface** dando in ingresso il vertice considerato. Si conteggiano i vertici che si trovano nel sottospazio positivo, negativo e sulla superficie. A partire da questo conteggio si determina la posizione dell'elemento rispetto alla superficie come mostrato nella tabella 2.1.

Nei casi dal 7 al 14 l'elemento è considerato tagliato, la *flag* dell'entità viene settata su CUTTED e la variabile **M_mapCutEle** viene aggiornata aggiungendo un *pair* contenente gli

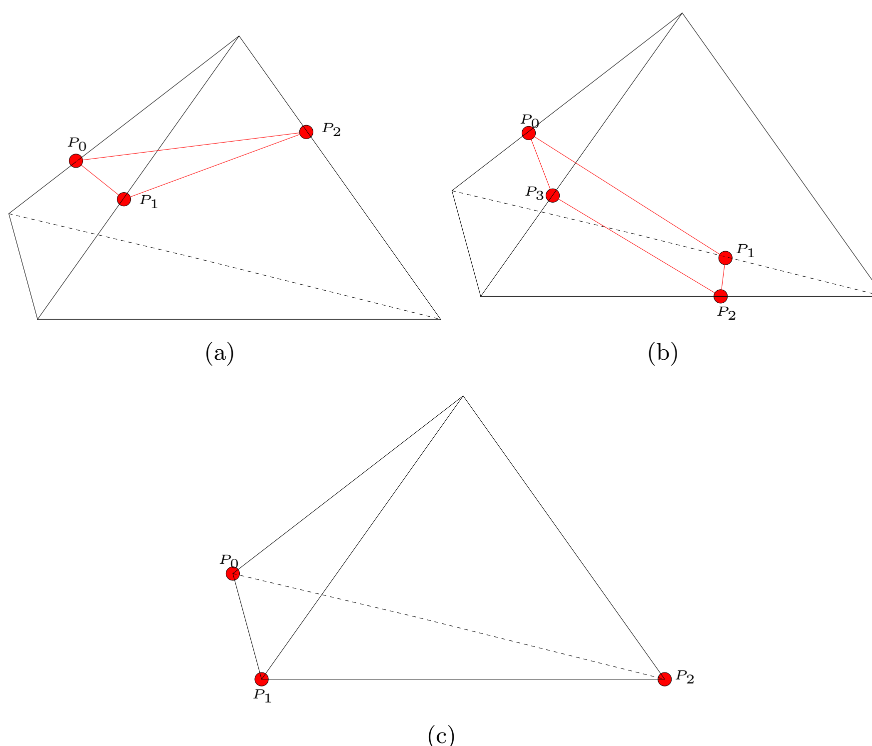


Figura 2.2: Taglio di un elemento dato un *level set*

ID della superficie e dell'elemento. Dopo aver definito se la superficie taglia l'elemento, viene aggiornata la variabile **M_region** nel *marker* (si veda 2.4.4).

Il taglio dell'elemento Il taglio dell'elemento implementato nel metodo **cutElement()** ha come parametri in ingresso l'ID dell'elemento tagliato e il numero **nbCut** di superfici che lo attraversano [11]. Indipendentemente dal numero di tagli attraverso l'elemento, questo metodo opera in diverse fasi che richiamano altri metodi della classe.
Inizialmente si calcolano le intersezioni tra l'elemento e le superfici (se più di una) tramite le funzioni **calculateIntersections()** e, quando **nbCut** è pari a 2, **calculateLevelSetIntersections()**; in seguito, a partire da questi nuovi punti e dai vertici del tetraedro, si costruiscono gli insiemi dei punti appartenenti ad una data parte dell'elemento (metodi **oneCut()** e **twoCut()** se **nbCut** è 2). L'inviluppo convesso di ciascun insieme viene calcolato e ritriangolato tramite `QHull` (implementazione nel metodo **buildBareEntities()**). I simplessi ottenuti sono poi associati ad una regione [12] e aggiunti alla variabile **M_subElements** nel *marker* in modo che ciascun elemento contenga l'informazione di come viene diviso dai *level* set che lo intersecano. Vediamo nel dettaglio come operano queste routine, iniziando dal caso di un elemento attraversato da un solo *level set*.

Consideriamo un elemento tagliato da una superficie definita dall'equazione $f(\vec{x}) = 0$. Cerchiamo i punti di intersezione tra i lati dell'elemento e la superficie, facendo attenzione al

[11] Il numero delle superfici che attraversano l'elemento è ottenuto attraverso il conteggio sulla chiave corrispondente all'ID dell'elemento nella mappa **M_mapCutEle**.

[12] Per regione intendiamo un sottospazio di $\mathbb{R}^3$ in cui il segno delle funzioni *level set* $f_i(\vec{x})$ rimane costante.

Tabella 2.1: Posizione di un elemento a partire dal segno della funzione *level set* nei vertici.

Caso	$f(\vec{x}) > 0$	$f(\vec{x}) < 0$	$f(\vec{x}) = 0$	
1	4	0	0	Elemento non tagliato
2	0	4	0	
3	3	0	1	
4	0	3	1	
5	2	0	2	
6	0	2	2	
7	3	1	0	Elemento tagliato
8	1	3	0	
9	1	2	1	
10	2	2	0	
11	2	1	1	
12	1	1	2	
13	1	0	3	Superficie passante per una faccia
14	0	1	3	

fatto che i vertici potrebbero essere anch'essi punti di intersezione (**calculateIntersections()**). L'algoritmo utilizzato è presentato nello pseudo-codice 1.
Facendo riferimento all'algoritmo 1, il calcolo dell'intersezione è fatto chiamando il metodo **intersection()** della superficie considerata, dando in ingresso i due estremi del lato P_1 e P_2. Il punto trovato viene aggiunto alla variabile **S_intersectionPoints** nel *marker* degli elementi. Viene inoltre aggiornata la mappa **M_newPointsOnEdges** e la variabile **M_pointsOnEdge**, quest'ultima nel *marker* del lato.

Quando un elemento è attraversato da due superfici, per definire la parti in cui esso viene diviso, non è sufficiente il calcolo delle intersezioni con ciascun *level set*.
In questo caso è necessario determinare anche i punti di intersezione tra le due superfici che si trovano sulle facce (si veda l'implementazione nel metodo **calculateLevelSetIntersections()**). Supponiamo infatti che due piani attraversino l'elemento, la loro intersezione è costituita da una retta che attraversa l'elemento da parte a parte e che interseca le facce in due punti (o al limite passano per un vertice). Possiamo pensare di generalizzare questo discorso a due superfici generiche. La curva data dalla loro intersezione può essere approssimata da un segmento congiungente i due punti di intersezione se la curvatura di entrambe le superfici è sufficientemente grande.
Questo segmento è comune a tutte le parti in cui l'elemento è diviso, ed è a partire da questo che verranno costruiti gli insiemi di punti (si veda la figura 2.3).
L'algoritmo di calcolo è presentato nello pseudo-codice 2, nel paragrafo successivo verrà invece spiegato nel dettaglio come utilizzare questi punti per la costruzione dei *sub element*.

Facendo riferimento all'algoritmo 2, per il calcolo dell'intersezione tra due segmenti si vuole risolvere il seguente sistema lineare, (dato dalla parametrizzazione dei due segmenti noti gli estremi):

Algoritmo 1 Calcolo dell'intersezione tra un *edge* e un *level set*

{Siano dati l'elemento k-esimo e la superficie j-esima}
for $i = 0 \rightarrow ne$ **do**
  {ciclo rispetto a tutte le ne *edge* dell'elemento k-esimo}
  p_1 è la posizione del primo vertice P_1 dell'*edge* rispetto alla superficie;
  p_2 è la posizione del secondo vertice P_2 dell'*edge* rispetto alla superficie;
  if $p_1 p_2 < 0$ **then**
    {L' *edge* è tagliato dal *level set*}
    if L'intersezione non è stata ancora calcolata **then**
      Calcola il punto di intersezione;
      if coincide con il punto (già calcolato) di intersezione di un'altra superficie con l'i-esima *edge* **then**
        Considera il punto già calcolato;
      else
        Aggiungi il nuovo punto;
      end if
    else
      Considera il punto già calcolato;
    end if
  end if
end for
Ritorna il vettore contenente gli ID di tutti i punti di intersezione;

$$\begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix} + t \begin{bmatrix} x_1 - x_0 \\ y_1 - y_0 \\ z_1 - z_0 \end{bmatrix} = \begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix} + m \begin{bmatrix} x_3 - x_2 \\ y_3 - y_2 \\ z_3 - z_2 \end{bmatrix}$$

dove $0 < t < 1$ e $0 < m < 1$.
Riscriviamo il sistema precedente nel seguente modo :

$$\begin{bmatrix} x_1 - x_0 & x_2 - x_3 \\ y_1 - y_0 & y_2 - y_3 \\ z_1 - z_0 & z_2 - z_3 \end{bmatrix} \begin{bmatrix} t \\ m \end{bmatrix} = \begin{bmatrix} x_2 - x_0 \\ y_2 - y_0 \\ z_2 - z_0 \end{bmatrix}$$

Il sistema è a tre equazioni per due incognite, quindi a prima vista si potrebbe pensare che questi punti non si trovino quasi mai. In realtà bisogna considerare il fatto che cerchiamo l'intersezione tra segmenti costruiti con punti su elementi che sappiamo essere tagliati da due superfici, per cui si tratta di casi in cui possiamo ragionevolmente aspettarci di trovare una soluzione: non verranno trovati punti di intersezione nel caso in cui le superfici che attraversano l'elemento non si intersecano tra loro, negli altri casi ne verranno trovate due a meno che esse non coincidano con punti già calcolati precedentemente.
I nuovi punti calcolati vengono aggiunti alla variabile **S_intersectionPoints** nel *marker* degli elementi.
Viene inoltre aggiornata la mappa **M_newPointsOnFaces** e la variabile **M_pointsOnFace**, quest'ultima nel *marker* della faccia su cui si trova il punto.

Algoritmo 2 Calcolo dell'intersezione tra due *level set* sulla faccia di un elemento

{Siano dati l'elemento k-esimo, i punti di intersezione $P_0^j, ..., P_{n_j}^j$ tra questo e la superficie j-esima e $P_0^m, ..., P_{n_m}^m$, punti di intersezione tra l'elemento e la superficie m-esima }
Costruisci tutti gli N_j possibili segmenti congiungenti i punti $P_0^j, ..., P_{n_j}^j$;
Costruisci tutti gli N_m possibili segmenti congiungenti i punti $P_0^m, ..., P_{n_m}^m$;
for $i_1 = 0 \to N_j$ **do**
  for $i_2 = 0 \to N_m$ **do**
    if Il segmento i_1-esimo e il segmento i_2-esimo non hanno un estremo in comune **then**
      Cerca l'intersezione tra il segmento i_1-esimo e il segmento i_2-esimo;
      if L'intersezione viene trovata **then**
        Determina la faccia a cui appartiene il punto
        if Il punto appartiene ad una delle facce dell'elemento k-esimo **then**
          if Il punto su tale faccia era già stato calcolato **then**
            Considera il punto già calcolato;
          else
            Il punto è aggiunto ai punti di intersezione;
          end if
        end if
      end if
    end if
  end for
end for
Ritorna il vettore contenente gli ID dei punti di intersezione;

Per determinare a quale faccia appartiene un punto, si cicla sulle facce dell'elemento, verificando se il nuovo punto è complanare con i vertici attraverso il metodo **controlVolumeArea()** (il volume del tetraedro costituito dai 4 punti deve essere inferiore alla tolleranza fissata).

Costruzione degli insiemi di punti Per il caso di taglio singolo, una volta determinate le intersezioni, costruiamo due nuvole di punti: in entrambe metteremo i punti di intersezione, in una aggiungeremo poi i vertici dell'elemento appartenenti al sottospazio in cui vale $f(\vec{x}) > 0$ nell'altra i restanti. L'elemento risulterà diviso in due parti definite dall'inviluppo convesso di queste nuvole di punti[13].
La costruzione di questi insiemi di punti è fatta dalla routine **oneCut()**[14].
Per il taglio doppio, la costruzione delle nuvole di punti è compiuta invece dal metodo **two-**

[13] Una piccola regione di sovrapposizione tra le due parti può formarsi qualora la superficie intersecante non sia un piano. Ipotizziamo che l'elemento sia intersecato da una sfera, i punti risultanti dalle intersezioni con i lati potrebbero non essere complanari. Questo si traduce nel fatto che una delle due parti in cui l'elemento viene diviso non sia convessa, ma poiché entrambe le parti vengono costruite come inviluppo convesso di una nuvola di punti, vi sarà una regione non nulla comune ad entrambe.

[14] Questa routine, dal momento che la costruzione delle nuvole di punti è analoga per elementi e facce tagliate, è pensata per poter calcolare sia i *sub-element* sie le *sub-face*, rappresentati entrambi da un *pair* contenente un vettore di *unsigned int*, i vertici, e un *usigned int* per la regione di appartenenza. Il parametro **dim** in ingresso indica la dimensione dell'entità che si sta tagliando (4 per un tetraedro e 3 per una faccia).

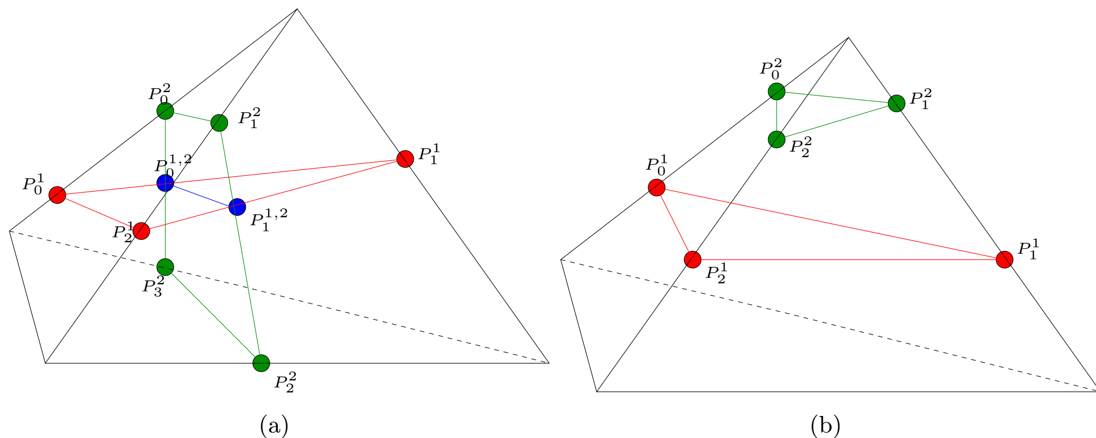


Figura 2.3: Taglio di un elemento dati due *level set*

Cut()[15] che in ingresso riceve i punti di intersezione tra elemento e superfici e i punti di intersezione tra superfici.

Facendo riferimento alla figura 2.3, dati due piani che intersecano l'elemento, esso può risultare diviso in tre o quattro parti (salvo casi degeneri in cui potrebbero essere due o una) perciò prevediamo 4 vettori contenenti gli ID dei punti. In tutti questi, per quanto già spiegato precedentemente, metteremo gli ID dei punti di intersezione tra le due superfici e le facce (punti i blu), poi nel vettore associato alla regione in cui entrambi i *level set* sono positivi metteremo gli ID di tutti i punti (tra vertici dell'elemento e intersezione lati-superficie) per i quali entrambe le funzioni *level set* sono maggiori o uguali a zero, e così via per gli altri vettori.
Noti questi insiemi di punti è il momento di calcolarne inviluppo convesso e triangolazione di Delaunay.

Triangolazione dei *sub-element* Dato un insieme di punti, si vuole ora ottenerne una triangolazione in modo da avere strutture geometriche su cui siano note delle formule di quadratura per risolvere numericamente il problema differenziale.
Per ritriangolare le parti in cui un elemento viene tagliato è stata utilizzata la routine di `QHull` [2] che calcola la triangolazione di Delaunay di un insieme di punti. E' stato dunque necessario integrare la libreria `QHull` in `LifeV`.
Nel file QhullWrapper.hpp è contenuta l'interfaccia di `QHull` per `LifeV`: date in ingresso le coordinate dei punti, ritorna un vettore contenenti i simplessi calcolati. All'interno della classe **XfemMeshHandler** è stato inserito un metodo **callDelaunayTriangulation()** che dati gli ID dei punti fornisce all'interfaccia di `QHull` le corrispondenti coordinate e una volta ottenuta la triangolazione riconverte l'output in vettori di ID di punti della *mesh* [16]. Questa routine ha anche il compito di proiettare i punti su un piano (xy, yz o xz) qualora si voglia triangolare una struttura 2D come un poligono come nel caso del calcolo delle *sub-face*. In questi casi `QHull` riceve in ingresso le coordinate della proiezione e le *sub-face* vengono poi ricostruite a partire dalla proiezione triangolata.
Se una nuvola di punti non è costituita da un numero sufficiente di punti (devono essere almeno

[15] Vale quanto detto in nota per la routine **oneCut()**.
[16] La routine di `QHull` assegna ai punti in input degli ID locali a partire da 0. I vettori in output contengono questi ID che vanno convertiti.

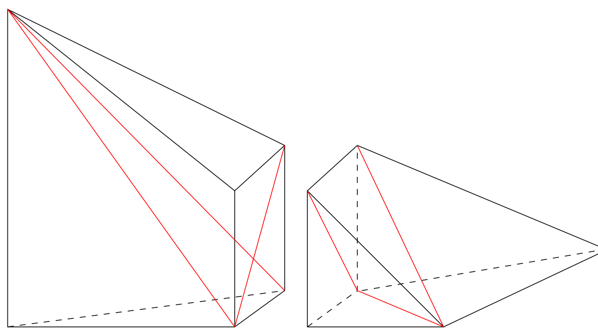


Figura 2.4: Elemento tagliato in cui le facce sono ritriangolate in modo non conforme

4 per avere un tetraedro) `QHull` non viene chiamato a triangolare. Se invece la lunghezza del vettore è superiore a 4 si procede alla triangolazione e i sotto-elementi calcolati, se non degeneri, vengono aggiunti al *marker*. Se il vettore contiene invece 4 ID, si controlla che non si tratti di un tetraedro degenere e poi, se non è quello il caso, viene creato un nuovo *sub element* e aggiunto anch'esso al *marker*. Quanto qui descritto è implementato nella routine **buildBareEntities()**.
L'uso di `QHull` ha posto due problemi. Innanzitutto le opzioni di `QHull` che impongono un output simpliciale (solo tetraedri) generano elementi a volume nullo, cioè costituiti da vertici complanari. Si è ovviato a questo problema calcolando di volta in volta il volume di tali elementi e inserendo solo quelli il cui volume supera una certa tolleranza definita dall'utente (**M_volumeTolerance**).
Il controllo sui volumi è fatto dalla routine **controlVolumeArea()** che dati 4 punti $P_0 = (x_0, y_0, z_0)^T$, $P_1 = (x_1, y_1, z_1)^T$,$P_2 = (x_2, y_2, z_2)^T$ e $P_3 = (x_3, y_3, z_3)^T$ e dato il parametro in input **dim** pari a 4, calcola il volume:

$$V = \frac{1}{6} \cdot \left| det \begin{bmatrix} x_1 - x_0 & y_1 - y_0 & z_1 - z_0 \\ x_2 - x_0 & y_2 - y_0 & z_2 - z_0 \\ x_3 - x_0 & y_3 - y_0 & z_3 - z_0 \end{bmatrix} \right|$$

Quando viene calcolato un *sub element* di volume nullo, la variabile **M_nullVolumes** viene incrementata.
Il controllo sulle aree è fatto anch'esso dalla routine **controlVolumeArea()** che dati 3 punti $P_0 = (x_0, y_0, z_0)^T$, $P_1 = (x_1, y_1, z_1)^T$,$P_2 = (x_2, y_2, z_2)^T$ e dato il parametro in input **dim** pari a 3, calcola l'area:

$$A = \frac{1}{2} \|(P_1 - P_0) \times (P_2 - P_0)\|$$

Il secondo problema sorto viene dal fatto che triangolare indipendentemente ciascuna parte degli elementi tagliati può portare ad avere triangolazioni differenti per una stessa *sub-face* comune (come illustrato in figura 2.4). Questo porta ad avere una *mesh* non conforme, e non permette di definire l'adiacenza di un elemento ad un altro.

Questa non conformità della ritriangolazione degli elementi tagliati non rappresenta tuttavia un problema. Difatti, essa serve a calcolare il contributo locale di ciascuna parte di elemento ma non costituisce un raffinamento della griglia in prossimità dell'interfaccia. I sotto-elementi non sono quindi nuovi elementi, con i propri nodi e gradi di libertà. Essi hanno il solo ruolo di permettere l'integrazione delle funzioni di base dell'elemento intero su parti di esso.

Peso dell'elemento Una volta che è nota la ritriangolazione dell'elemento date le superfici che lo intersecano è possibile calcolare e attribuire il peso dell'elemento chiamando la routine **setWeight()** del **MarkerElementXfem**.

2.4.3 Il taglio delle facce

Prima di analizzare gli algoritmi, è necessario distinguere due tipi di facce che vengono prodotte dal taglio di un elemento: con facce 'di taglio' (*cutting face*) intendiamo quelle parti di superficie [17] che, intersecando un elemento, lo tagliano in due parti, diverse dalle facce tagliate che non sono altro che ritriangolazioni delle facce di un elemento tagliato.

Facce 'di taglio' Come è stato mostrato nel capitolo precedente, nella formulazione debole sono presenti alcuni termini di interfaccia. Per il calcolo di questi, è necessario conoscere una triangolazione dell'interfaccia Γ all'interno di ogni elemento tagliato. La superficie Γ_K (o meglio, la sua approssimazione $\Gamma_{K,h}$) viene triangolata in ogni elemento K tagliato, a partire dai punti di intersezione tra Γ e i lati dell'elemento. Questa triangolazione viene utilizzata sia per il calcolo dei contributi di K', sia per quelli dell'elemento K''.
Per il calcolo di questa triangolazione, come si è detto, è necessario conoscere i punti di intersezione delle superfici con i lati dell'elemento (vertici compresi) più quelli eventualmente calcolati dalla routine **calculateLevelSetIntersections()**.
Se la superficie di intersezione è una sola, il vettore contenente gli ID dei punti di intersezione (vertici dell'elemento compresi) viene dato in input alla routine **callDelaunayTriangulation()** in modo da ottenere la ritriangolazione della superficie che attraversa l'elemento. Altrimenti, nel caso di due superfici intersecanti i e j, si costruiscono i vettori di punti nel seguente modo (si veda il metodo **cuttingElementFaces()** che viene chiamato da **cutElement()**):

- punti tali che $f_i(\vec{x}) = 0$ e $f_j(\vec{x}) \geq 0$;
- punti tali che $f_i(\vec{x}) = 0$ e $f_j(\vec{x}) \leq 0$;
- punti tali che $f_j(\vec{x}) = 0$ e $f_i(\vec{x}) \geq 0$;
- punti tali che $f_j(\vec{x}) = 0$ e $f_i(\vec{x}) \leq 0$.

Infine, nel metodo **buildAndAddCuttingFaces()**, questi insiemi di punti vengono ritriangolati (se presenti almeno 3 vertici) e le *cutting face* vengono aggiunte al *marker* dell'elemento (variabile **M_cuttingFaces**).
Durante la fase di triangolazione viene calcolata l'area di ogni *cutting face* e la somma di tali aree, divise per regione, viene memorizzata nella variabile **M_cuttingFacesArea**. Il contenuto di tale variabile è parte dell'output della routine **intersectMeshSurfaces()** in quanto permette all'utente di avere un primo controllo sulla correttezza della triangolazione della superficie.

Facce dell'elemento Questa parte non è ancora stata implementata anche se tutte le strutture dati necessarie per conservare le informazioni sui tagli delle facce degli elementi, necessarie in problemi iperbolici per il calcolo dei flussi attraverso gli elementi tagliati, sono già presenti nel codice (si veda la sezione 2.3.1).

[17] Si intende sempre la curva di livello nulla della funzione *level set*

Anche l'algoritmo di taglio di una faccia è già stato implementato, in quanto è il medesimo utilizzato nel taglio di un elemento, nei metodi **oneCut()**, **twoCut()** e **buildBareEntities()**. Tali metodi operano nel medesimo modo per facce (quindi triangoli) e elementi (tetraedri), ciò che cambia sono i punti in ingresso. E' questa appunto la parte da implementare.
E' necessaria una routine che data una faccia cerchi gli ID dei punti di intersezione sui lati, l'eventuale punto di intersezione tra superfici [18] e determini il segno della funzione *level set* nei vertici. Date queste informazioni, il calcolo del taglio è analogo a quanto fatto per l'elemento.

2.4.4 Definizione della regione di appartenenza

Si è cercato un modo per definire in modo univoco la regione di appartenenza di un elemento rispetto alle superfici che intersecano la *mesh* [19]. Il metodo che verrà presentato è implementato nella routine **updateElmentRegion()** della classe **XfemMeshHandler**.
Consideriamo prima una sola superficie. Attribuiamo 0 se l'elemento si trova nella regione per cui vale $f(\vec{x}) > 0$, 2 se si trova nella regione tale che $f(\vec{x}) < 0$ e 1 se l'elemento è tagliato quindi attraversato dalla superficie definita da $f(\vec{x}) = 0$ [20].
Con due superfici, applicando il medesimo ragionamento e considerando tutte le possibili combinazioni, otteniamo la tabella 2.2.

Tabella 2.2: Assegnazione della *flag* **M_region** di un elemento in una *mesh* attraversata da due superfici.

	$f_1(\vec{x}) > 0$	$f_1(\vec{x}) = 0$	$f_1(\vec{x}) < 0$
$f_2(\vec{x}) > 0$	0	1	2
$f_2(\vec{x}) = 0$	3	4	5
$f_2(\vec{x}) < 0$	6	7	8

Per fare un esempio, un elemento che rispetto alla seconda superficie si trova nella regione negativa ed è tagliato dalla prima superficie, apparterrà alla regione 7.

Generalizzando, per attribuire ad un elemento la regione, note la posizione dell'elemento rispetto a tutte le n superfici, ciclando su $i = 0, ..., n$ si aggiunge 3^i se è tagliato dalla superficie i-esima mentre si aggiunge $2 \cdot 3^i$ se si trova nel sottospazio in cui $f_i(\vec{x}) < 0$. Se si trova nel sottospazio in cui $f_i(\vec{x}) > 0$ la variabile **M_region** non viene incrementata.
Tale convenzione è quindi estendibile ad un numero arbitrario di superfici intersecanti l'elemento ed è equivalente a formare una cifra in base 3.
Data la regione è possibile inoltre tornare indietro e determinare le posizioni dell'elemento rispetto alle superfici contenute nel **SurfaceSet** tramite questo l'algoritmo 3.
Questa scelta permette anche di ridefinire facilmente la regione di ciascun *sub-element* a partire dalla variabile **M_region** dell'elemento tagliato. Consideriamo prima il caso di un singolo taglio. Sia i l'ID della superficie che taglia l'elemento, la regione dei sub-element che si trovano al di sopra [21] sarà data dalla regione dell'elemento tagliato meno 3^i, mentre per gli altri *sub-*

[18] Tali punti sono già calcolati e inseriti nella variabile **S_intersectionPoints**.
[19] Si è inoltre fatto attenzione a ideare un modo che potesse avere un senso anche per definire l'appartenenza di una *sub-face* o di un elemento tagliato ad una certa regione.
[20] Facendo riferimento alla tabella 2.1 attribuiamo 0 nei casi 1-3-5, 2 nei casi 2-4-6 e 1 nei restanti
[21] Dove vale la relazione $f_i(\vec{x}) > 0$

Algoritmo 3 Posizione dell'elemento data la *flag* **M_region**

a = M_region;
for $i = n \to 0$ **do**
  {Ciclo rispetto a tutte le n superfici}
  $q = \lfloor a/3^i \rfloor$;
  if $q = 0$ **then**
    Rispetto alla superficie i-esima si trova nel sottospazio positivo
  end if
  if $q = 1$ **then**
    E' tagliato dalla superficie i-esima
  end if
  if $q = 2$ **then**
    Rispetto alla superficie i-esima si trova nel sottospazio negativo
  end if
  $a \leftarrow a - q \cdot 3^i$
end for

element si dovrà sommare 3^i.Questo si generalizza facilmente al caso del taglio doppio. Per quanto riguarda le facce, la convenzione per l'attribuzione della regione è la medesima.

Capitolo 3

Integrazione del problema differenziale

3.1 Le strutture dati presenti in `LifeV`

In questo capitolo viene presentata quella parte di implementazione necessaria per risolvere il problema discreto: dalla gestione del parallelismo all'esportazione e visualizzazione della soluzione, passando per l'assegnazione dei gradi di libertà sullo spazio arricchito e la costruzione e assemblaggio del sistema lineare.
Per comprendere l'implementazione proposta è necessario avere un'idea generale sulle strutture dati che in `LifeV` si occupano di questi aspetti:

- **MeshPartitioner**, classe che partiziona la *mesh* sfruttando l'algoritmo di partizionamento *k-way*[1] applicato al grafo associato alla griglia, implementato nella libreria `ParMetis`[17] (ulteriori dettagli saranno forniti nella sezione 3.2);
- classi che definiscono lo spazio ad elementi finiti e i gradi di libertà: **FESpace** e **DOF**. Al momento dell'istanziazione della prima, viene costruito anche un oggetto di classe **DOF** in cui è presente una mappa che permette di conoscere i gradi di libertà associati a ciascun elemento. Queste classi e la mappa dei gradi di libertà sono definite localmente su ciascun processore. Per permettere la comunicazione tra processori in fase di assemblaggio della matrice, il calcolo della soluzione e l'esportazione, viene generata, a partire dalle mappe locali, una mappa `Epetra` [16]. Tramite essa è possibile conoscere su quali partizioni e quindi su quali processori si trova un dato grado di libertà. Questa mappa è essenziale per una corretta costruzione della matrice e vettore globale. Date le informazioni nella mappa `Epetra` è possibile infatti assemblare i contributi relativi ad un grado di libertà su un nodo che si trova sulla frontiera tra due partizioni.
  Nella classe **FESpace** sono presenti informazioni circa lo spazio di approssimazione (grado dei polinomi), l'elemento di riferimento (**ReferenceFE**), le formule di quadratura (**QuadratureRule**) sugli elementi e sulle entità $d-1$ dimensionali e alcune routine di interpolazione di funzioni sullo spazio;

[1] Il problema di partizionamento *k-way* è definito come segue. Consideriamo un grafo $G = (V, E)$ dove V è l'insieme di n nodi e E l'insieme dei lati che connettono i nodi. La misura di V nel caso più generale è data dalla somma dei pesi dei nodi presenti. Si vuole partizionare il grafo in k sottoinsiemi, $V_1, ..., V_k$ tali che $V_i \cap V_j = \emptyset$ per $i \neq j$, $\mid V_i \mid = \mid V \mid /k$ e $\bigcup_i V_i = V$ e il numero di lati di E i cui estremi appartengono a differenti sottoinsiemi è minimizzato.

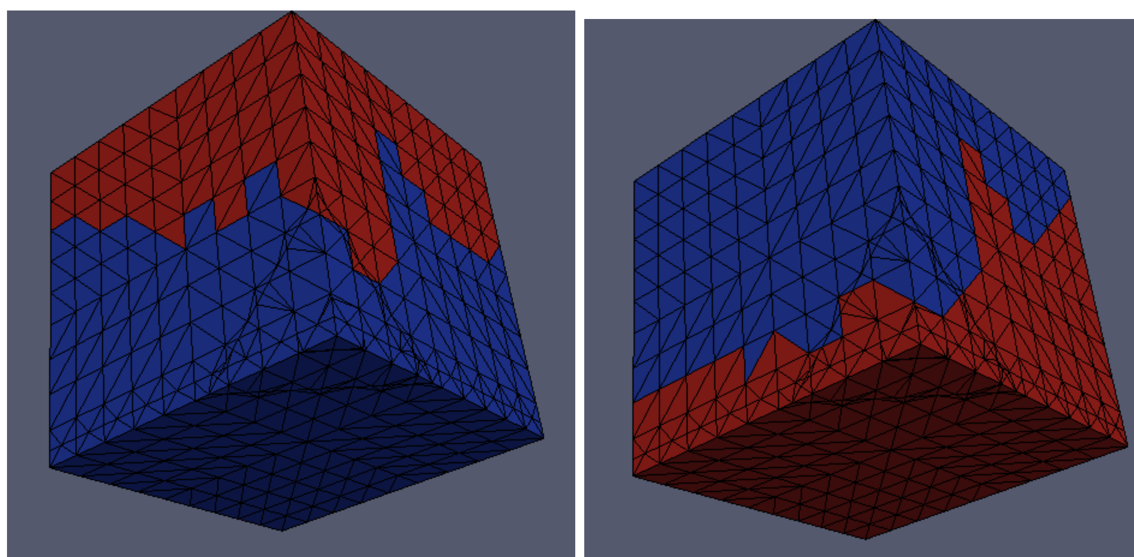
- **CurrentFE** e **CurrentBoundaryFE**, classi che permettono di calcolare i valori delle funzioni di base e di loro derivate nei nodi di quadratura a partire dall'elemento di riferimento;
- classi che assemblano le matrici ricavate dalla discretizzazione del problema differenziale e risolvono il sistema. A seconda del problema in esame sono state implementate classi diverse. Per questa tesi si è preso spunto dalla classe **ADRAssembler** che assembla il sistema lineare per un problema di diffusione-trasporto-reazione;
- classi **Exporter** e derivate che permettono di esportare la soluzione in un formato leggibile da programmi come `ParaView` [15] per il *rendering* e il *post-processing*.

Nel *namespace* **Assembly Elemental** sono contenute le routine che permettono di calcolare le matrici locali (*stiffness*, massa, stabilizzazioni, etc.).

3.2 Parallelismo e partizionamento

Si è già mostrato che in un calcolo ad Elementi Finiti Estesi (XFEM) sugli elementi attraversati dalla superfici di discontinuità è necessario aggiungere gradi di libertà.
Consideriamo infatti un elemento tetraedrico tagliato in due da una superficie: per poter avere soluzione discontinua è necessario raddoppiare i gradi di libertà così da poter integrare sulle due parti dell'elemento e avere una soluzione diversa per ciascuna di esse .
Il discorso si complica se più superfici attraversano l'elemento, difatti se le interfacce non si intersecano all'interno di K l'elemento risulterà diviso in tre parti mentre in caso di intersezione non nulla avremo quattro parti. Nel primo caso dovremo triplicare i gradi di libertà dell'elemento, nel secondo dovremo quadruplicarli [8].
L'aumento dei gradi di libertà è quindi legato a come l'elemento viene tagliato dalle superfici.
Questo aumento dei gradi di libertà sul singolo elemento comporta il fatto che su alcuni elementi il calcolo sarà maggiormente oneroso dovendo integrare separatamente su ciascuna parte: potremmo dire che alcuni elementi pesano più di altri.
Il partizionamento *k-way* operato dalla classe **MeshPartitioner** che richiama l'opportuna routine di `ParMetis`[17] considera in egual modo tutti gli elementi e produce quindi una partizione in cui ciascun processore ha grosso modo lo stesso numero di elementi. Nel caso XFEM questo può condurre ad avere una partizione non equilibrata se, ad esempio, la maggior parte degli elementi tagliati viene a trovarsi su un medesimo processore, con il risultato che alcuni processi potrebbero essere sensibilmente più lenti di altri.
Per ovviare a questo problema si è introdotta la variabile **M_nodeWeights**, un vettore di interi contenenti i pesi degli elementi della *mesh*, e il metodo **setNodeWeights()**. Il termine *node* è dovuto al fatto che `ParMetis` partiziona il grafo legato alla *mesh* in cui i nodi rappresentano gli elementi e i lati costituiscono le facce e quindi le adiacenze tra i diversi elementi.
In un calcolo XFEM è dunque necessario modificare questa variabile prima di lanciare il partizionamento inserendo i pesi degli elementi della griglia.
Prima della chiamata di `ParMetis` viene controllata la lunghezza del vettore:

- se è nulla, la chiamata della routine di partizionamento del grafo è quella standard;



(a) Non pesato (b) Pesato

Figura 3.1: Confronto tra partizionamento pesato e non pesato

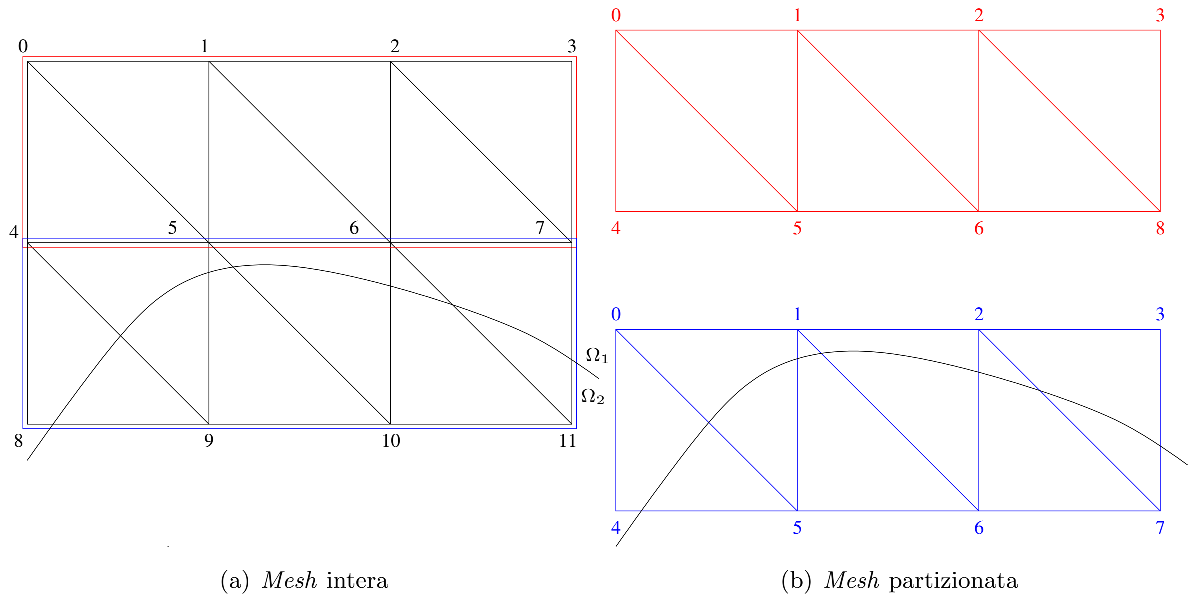
- se è non nulla, il vettore dei pesi è passato a `ParMetis` in modo che produca una partizione bilanciata dal punto di vista del carico computazionale.

E' stato implementato il metodo **createMeshPartition()** nella classe **XfemMeshHandler** che permette di scegliere tra partizionamento standard e partizionamento pesato.
Il partizionamento della *mesh* cambia dunque a seconda che venga considerato o meno il peso degli elementi. Se il partizionatore non ha informazioni sul peso degli elementi c'è il rischio di avere concentrazioni di elementi tagliati solo su alcuni processori come in figura 3.1, fatto che potrebbe compromettere l'efficienza di un eventuale calcolo in parallelo.

3.3 Assegnazione dei gradi di libertà

La gestione dei gradi di libertà deve ovviamente essere modificata rispetto al caso standard. E' stata pertanto implementata la classe **DOFXfem** in cui l'assegnazione dei gradi di libertà tiene conto del fatto che sugli elementi tagliati il loro numero è diverso da quello di un elemento finito standard.
L'attribuzione dei gradi di libertà in `LifeV`, implementata nella classe **DOF**, è basata sull'ID globale dell'entità associata al nodo: nel caso di elementi finiti P1, ad esempio, ad ogni vertice della *mesh* è associato un grado di libertà il cui ID coinciderà con l'ID globale del nodo; nel caso P0 è associato un grado di libertà ad ogni elemento e anche in questo caso i due ID coincideranno.
Il fatto di basarsi sull'ID globale dell'entità geometrica è la chiave per consentire la parallelizzazione del codice in quanto ogni partizione costruisce la propria mappa di gradi di libertà preservando il fatto che ad uno stesso nodo, su processori diversi, corrisponde il medesimo grado di libertà[2].

[2]La mappa dei gradi di libertà **M_localToGlobal** è una sorta di matrice a cui si accede in base all' ID locale dell'elemento e il numero locale (cioè sull'elemento) del grado di libertà. Il metodo **localToGlobalMap()** della



(a) *Mesh* intera (b) *Mesh* partizionata

Figura 3.2: Esempio di *mesh* partizionata

Nel caso di un calcolo XFEM secondo l'approccio in [14], il fatto di aumentare i gradi di libertà su alcuni elementi rende più difficile la costruzione della mappa. Consideriamo infatti il caso di un elemento finito P1 attraversato da Γ che si trova sulla frontiera tra due partizioni. Al momento di assegnare i gradi di libertà ai nodi della faccia sulla frontiera si pone il problema di come attribuire gli stessi gradi di libertà raddoppiati all'elemento adiacente che si trova su un altro processore sapendo che attraverso l'ID globale è già stato assegnato il primo grado di libertà per nodo.

Questo è il caso dei nodi 4-11 della *mesh* in la figura 3.2. Quanto implementato attualmente nella classe **DOF** assegnerebbe a questi nodi rispettivamente i gradi di libertà 4-11; ma non si tiene conto che su questi nodi c'è più di un grado di libertà[3].
La soluzione in cui si assegna al grado di libertà aggiuntivo il numero dato dai gradi totali standard sui nodi + 1 (se è il primo nodo da raddoppiare che viene incontrato) diventa inefficace quando il calcolo è svolto in parallelo. Allo stesso nodo raddoppiato difficilmente sarà assegnato il medesimo grado di libertà da processori diversi, che ciclano sugli elementi della propria partizione.
A questo punto si sono valutate due possibili soluzioni:

- assegnare i gradi di libertà in modo sequenziale comunicando di volta in volta l'avvenuta aggiunta a tutti i processori;
- assegnare a monte del partizionamento degli ID virtuali globali ad ogni entità su cui sono presenti gradi di libertà (nel caso P1, solo i vertici degli elementi) legati alla *flag* della regione dell'elemento.

classe **DOF**, dati questi due parametri in ingresso, restituisce il numero di grado di libertà globale, cioè quello necessario per assemblare la matrice.

[3]E' previsto che su uno stesso nodo possano essere presenti più di un grado di libertà ma questo dovrebbe essere un fatto generale, comune anche a tutti gli altri nodi. Non è previsto che l'aumento dei gradi di libertà di un nodo sia solo locale.

Si è deciso di implementare questa seconda strada, che permette di gestire in modo abbastanza efficiente anche il caso di tagli multipli, in cui un elemento venga tagliato da due *level set* e non è più sufficiente raddoppiare i gradi di libertà.

3.3.1 Implementazione: le classi MultipleDOFHandler e DOFXfem

L'idea è quella di avere una classe che, data una *mesh* contenente le informazioni sull'intersezione con i *level set* presentate nel capitolo precedente, assegna ad ogni entità su cui sono presenti gradi di libertà tanti ID virtuali globali quante sono le regioni presenti negli elementi a cui esse appartengono. A partire da questi ID globali virtuali verranno assegnati i gradi di libertà. Il fatto che questi ID hanno valore globale evita i problemi legati al parallelismo presentati nella sezione precedente.
Vediamo nel dettaglio le classi implementate e come operano le routine di assegnazione dei gradi di libertà.

La classe MarkerXFEM Questa classe, da cui le classi **MeshElementMarkedxD** ereditano, è un contenitore per gli ID globali virtuali di un entità, riferiti ad una specifica regione. All'interno di questa classe troviamo i seguenti attributi:

- **M_entityDofMap**, mappa della STL in cui il primo *unsigned int* è la *flag* relativa alla regione in cui si trova l'entità e il secondo *unsigned int* è l'ID globale virtuale dell'entità nella regione;

- **S_nbGlobalVirtualVertices**, **S_nbGlobalVirtualEdges**, **S_nbGlobalVirtualFaces** e **S_nbGlobalVirtualVolumes**, variabili statiche intere che conteggiano il numero di entità virtuali[4].

L'assegnazione di queste variabili è compito di una routine della classe **MultipleDOFHandler** che viene ora presentata.

Le classe MultipleDOFHandler Come la classe **XfemMeshHandler** è un contenitore di routine volte a ottenere informazioni geometriche che poi inserisce negli opportuni *marker*, così questa classe si occupa di arricchire la *mesh* con le informazioni necessarie ad una corretta assegnazione dei gradi di libertà.
E' dato un solo costruttore che chiede come parametro una referenza al **DOFLocalPattern** dello spazio di approssimazione (a livello implementativo si tratta di un oggetto di classe **FESpace**) su cui si si cerca la soluzione. Se i **DOFLocalPattern** non coincidono si genereranno errori al momento della creazione della mappa dei gradi di libertà in **DOFXfem**.
Prima di partizionare la matrice è necessario eseguire la routine **updateDOFMaps()** dando come parametro in ingresso uno *smart pointer* alla *mesh* intera. Questa routine riempie le mappe contenute nei *marker* presentate al paragrafo precedente e aggiorna gli attributi statici di **MarkerXfem**. Nello pseudo-codice 4 è mostrato come vengono assegnati gli ID virtuali globali alle entità su cui sono presenti gradi di libertà. In particolare viene mostrato il caso di gradi di libertà posizionati sui vertici di un elemento.

A questo punto è possibile partizionare la *mesh* e costruire gli spazi opportuni.

[4]Intendiamo il numero di entità geometriche pesato per il numero di regioni in cui esse devono essere rappresentate.

Algoritmo 4 Assegnazione degli ID virtuali globali alle entità associate ai gradi di libertà

$vCount \leftarrow 0$
for $ie = 0 \rightarrow ne$ **do**
  {ciclo su tutti gli elementi}
  for $ir = 0 \rightarrow nr$ **do**
    {ciclo sulle regioni in cui l'elemento è diviso}
    if gradi di libertà associati al nodo > 0 **then**
      for $iv = 0 \rightarrow nv$ **do**
        {ciclo sui vertici dell'elemento}
        if la regione non è presente nella mappa di d.o.f. nel *marker* del punto **then**
          {ID virtuale globale del punto associato alla regione ir non è stato assegnato}
          Aggiungi l'ID virtuale del punto $vCount$ associato alla regione ir come *pair* $(ir, vCount)$ nella mappa del *marker*;
          $vCount \leftarrow vCount + 1$;
        end if
      end for
    end if
  end for
end for

La classe DOFXfem Questa classe è alla base dell'implementazione di uno spazio di elementi finiti arricchito. Eredita pubblicamente dalla classe **DOF** e in essa vengono aggiunti i seguenti attributi:

- **M_nbVirtualGlobalElements**, variabile intera contenente il numero di elementi virtuali globali;
- **M_nbVirtualLocalElements**, variabile intera contenente il numero di elementi virtuali locali (cioè sulla partizione considerata);
- **M_cutElementDOF**, mappa della *standard library.*

Quest'ultima variabile permette di gestire il fatto che su ciascuna partizione il numero di elementi geometrici presenti non corrisponde al numero di elementi virtuali locali da cui dipende il numero di gradi di libertà: ad un elemento geometrico corrispondono tanti elementi virtuali quante le regioni in cui esse deve essere rappresentato.
La mappa **M_localToGlobal** contenuta in **DOFXfem** deve prevedere quindi un numero di colonne (gli elementi) pari al valore di **M_nbVirtualLocalElements** diversamente dalla mappa della classe **DOF** che aveva un numero di colonne pari al numero di elementi geometrici nella partizione. L'accesso alla mappa **M_localToGlobal** nella classe **DOFXfem** sarà fatto non più attraverso l'ID locale dell'elemento ma attraverso un ID locale virtuale, cioè un ID che viene attribuito agli elementi in modo sequenziale (su ciascuna partizione), uno per ogni regione in cui l'elemento deve essere rappresentato. **M_cutElementDOF** permette di ottenere, dato l'ID locale di un elemento e la regione in cui esso deve essere rappresentato, l'ID locale

virtuale dell'elemento, in modo da poter conoscere i suoi gradi di libertà chiamando la routine **localToGlobalMap()**.
La scelta di costruire uno spazio di elementi finiti a partire da **DOFXfem** è stata fatta aggiungendo un costruttore alla classe **FESpace**. Tramite questo costruttore è possibile specificare la *flag* **typeDOF**: 0 istanzia un oggetto di classe **DOF**, 1 per usare le funzionalità della classe **DOFXfem** e 2 per uno spazio discontinuo (si veda 3.5.2). Se **typeDOF** ha valore 1, viene eseguita la routine **updateXfemDof()**[5] che costruisce le mappe **M_cutElementDOF** e **M_localToGlobal**. Quest'ultima viene costruita assegnando i gradi di libertà come viene mostrato nello pseudo-codice 5. E' importante fare alcune osservazioni a questo punto sulle

Algoritmo 5 Assegnazione dei gradi di libertà sui vertici degli elementi in uno spazio arricchito

$gCount \leftarrow 0$;
if gradi di libertà associati ai vertici > 0 **then**
  for $ie = 0 \rightarrow ne$ **do**
    {ciclo sugli elementi della partizione}
    for Primo elemento virtuale → Ultimo elemento virtuale **do**
      {ciclo sugli elementi virtuali associati allo *ie*-esimo elemento geometrico}
      $lc \leftarrow 0$;
      for $iv = 0 \rightarrow nv$ **do**
        {ciclo sui vertici dell'elemento}
        for $i = 0 \rightarrow nd$ **do**
          {*nd* è il numero di gradi di libertà per ogni vertice}
          $id \leftarrow$ l'id virtuale globale del vertice per la regione dell'elemento virtuale considerato;
          Assegna all'*lc*-esimo grado di libertà locale dell'elemento virtuale il grado di libertà globale $gCount + id * nd + i$;

          $lc \leftarrow lc + 1$;
        end for
      end for
    end for
  end for
end if

differenze tra la classe **DOF** e la classe **DOFXfem**:

1. nella classe **DOFXfem** l'accesso alla mappa dei gradi di libertà deve essere fatto attraverso l'ID locale virtuale dell'elemento e il numero locale del grado di libertà. Per ottenere il primo di questi si chiama il metodo **getVirtualEle()** che interroga la mappa **M_cutElementDOF** dato ID locale dell'elemento geometrico e regione;

2. è stato fatto l'*overridding* del metodo **numElements()** che ritorna il numero di elementi locali virtuali.

[5]Il nome è differenta dall'analoga routine presente in **DOF** per l'impossibilità di *virtualizzare* un metodo *template*.

E'stato inoltre aggiunto un metodo **showMeByMesh()**, più completo rispetto a quello già implementato in quanto fornisce informazioni sugli ID locali,globali e virtuali degli elementi.

3.4 Risoluzione del problema discreto

3.4.1 Integrazione su un elemento tagliato

In un codice di calcolo ad Elementi Finiti è necessario disporre di tecniche di integrazione numerica sufficientemente accurate per il calcolo degli integrali presenti nella formulazione debole [20]. Sono quindi usate delle formule di quadratura che, in genere, presentano la seguente forma:

$$\int_K f(\mathbf{x})d\mathbf{x} \approx \sum_{q=1}^{m} f(\mathbf{x}_q)w_q$$

dove K indica la regione su cui si integra (un elemento della griglia), m è il numero di nodi di quadratura per la formula scelta, $\mathbf{x}_q$ i di quadratura e w_q sono i pesi.
Le formule di quadratura gaussiane sono quelle utilizzate nel calcolo degli integrali in `LifeV`. Il calcolo è svolto sull'elemento di riferimento, sul quale è nota l'espressione delle funzioni di base, mediante un opportuno cambio di variabile. Indichiamo con $\hat{x}_i$ le coordinate nello spazio di riferimento e con $\hat{\phi}_i$ le relative funzioni di base e con J la matrice Jacobiana della trasformazione geometrica tra elemento K ed elemento di riferimento:

$$J = [J_{ij}] = \left[\frac{\partial x_i}{\partial \hat{x}_i}\right]$$

Pertanto si ottiene:

$$\int_K f(\mathbf{x})d\mathbf{x} = \int_{\hat{K}} \hat{f}(\hat{\mathbf{x}}) \mid J(\hat{\mathbf{x}}) \mid d\hat{\mathbf{x}} \approx \sum_{q=1}^{m} \hat{f}(\hat{\mathbf{x}}_q) \mid J(\hat{\mathbf{x}}_q) \mid w_q$$

dove $\mid J \mid$ indica il determinante di J.
Nel metodo XFEM presentato in [14] è necessario sapere integrare su parti di un elemento e pertanto le formule di quadratura presentate devono essere modificate.

Dato un elemento K intersecato da Γ, supponiamo di voler integrare su ciascuna parte $K_i = K \cap \Omega_i$. Supponiamo di aver ritriangolato K_i in n_i sotto-elementi $K_i^1, ..., K_i^{n_i}$ come mostrato in 2.4.2.
Avremo che :

$$\int_{K_i} f(\mathbf{x})d\mathbf{x} = \sum_{j=1}^{n_i} \int_{K_i^j} f(\mathbf{x})d\mathbf{x}$$

L'integrale su una parte di elemento è quindi la somma dei contributi di tutti i sotto-elementi di una data regione.
Consideriamo a titolo di esempio il calcolo della matrice di massa locale: si tratta di integrare il prodotto della funzioni di base di un elemento K su tutti i sotto-elementi K_i^j. Per l'integrale è necessario dunque valutare il valore delle funzioni di base ϕ (dell'elemento intero K) nei nodi di quadratura del sotto-elemento K_i^j.
Lo *splitting* dell'elemento K in K' e K'' comporterà poi un assemblaggio differente dei contributi locali all'interno della matrice globale. Il contributo dei sotto-elementi K_1^j dovrà essere

assemblato coerentemente con i gradi di libertà dell'elemento K' mentre il contributo dei K_2^j sarà assemblato rispetto ai gradi di libertà di K''.
Il calcolo della matrice di massa per l'elemento K' deve quindi essere fatto nel seguente modo[6]. Siano $\mathbf{x}_q^l$ i nodi di quadratura sul sotto-elemento K_1^l.

$$\begin{aligned}\int_{K_1} \phi'_\alpha(\mathbf{x})\phi'_\beta(\mathbf{x})d\mathbf{x} &= \sum_{l=1}^{n_1} \int_{K_1^l} \phi'_\alpha(\mathbf{x})\phi'_\beta(\mathbf{x})d\mathbf{x} \\ &= \sum_{l=1}^{n_1} \int_{\hat{K}_1^l} \phi'_\alpha(\hat{\mathbf{x}})\phi'_\beta(\hat{\mathbf{x}}) \mid J_l(\hat{\mathbf{x}}) \mid d\hat{\mathbf{x}} \\ &\approx \sum_{j=1}^{n_1} \sum_{q=1}^{m} \phi'_\alpha(\hat{\mathbf{x}}_q^l)\phi'_\beta(\hat{\mathbf{x}}_{iq}^l) \mid J_l(\hat{\mathbf{x}}_q^l) \mid w_q^l\end{aligned}$$

dove, è importante ricordarlo, le ϕ' non sono le funzioni di base dei sotto-elementi K_1^j ma quelle dell'elemento K, attraversato dalla discontinuità.
Con questo approccio è possibile costruire tutti i contributi locali e costruire il sistema associato al problema discreto 1.7.

3.4.2 Costruzione del sistema lineare: la classe XfemAssembler

Richiamiamo innanzitutto il problema discreto che vogliamo risolvere. Trovare $U \in V^h$ tale che:

$$a_h(U, \phi) = L(\phi), \forall \phi \in V^h$$

dove la forma bilineare è composta dai seguenti termini:

$$a_h(U,\phi) = \underbrace{(\alpha_i \nabla U_i, \nabla \phi_i)_{\Omega_1 \cup \Omega_2}}_{I} \underbrace{-([U], \{\alpha \nabla_n \phi\})_\Gamma - (\{\alpha \nabla_n U\}, [\phi])_\Gamma}_{II} \underbrace{+(\lambda [U], [\phi])_\Gamma}_{III} \tag{3.1}$$

e il funzionale L è definito come:

$$L(\phi) = \underbrace{(f, \phi)_\Omega}_{III} \underbrace{+(\kappa_2 g, \phi_1)_\Gamma + (\kappa_1 g, \phi_2)_\Gamma}_{IV} \tag{3.2}$$

In `LifeV` è stata implementata la classe **XfemAssembler** che ha il compito di calcolare i contributi locali di ogni elemento e di assemblare la matrice globale.

Essa contiene i seguenti metodi:

- **addDiffusion()**, che costruisce la matrice globale di *stiffness* (primo termine della 3.1) a partire dalle matrici locali di tutti gli elementi della triangolazione;
- **addJumpOnSolution()**, che costruisce la matrice associata al secondo termine della 3.1 integrando sulle *cutting face* di ogni elemento $K \in \mathcal{G}_h$;

[6] A livello di notazione, gli apici di ϕ indicano l'elemento K' o K''.

- **addPenalty()**, che costruisce la matrice associata al termine di penalità (terzo termine della 3.1) attraverso il quale nel caso implementato si impone la continuità della soluzione attraverso Γ, integrando sulle *cutting face* di ogni elemento $K \in \mathcal{G}_h$;
- **addMassRhs()**, che assembla il vettore della forzante (primo termine in 3.2);
- **addJumpRhs()**, che assembla il vettore dovuto all'imposizione del salto sulla derivata conormale.

Analizziamo ora nel dettaglio queste funzioni.

Matrice di *stiffness* Per costruire la matrice di *stiffness*, si effettua prima un ciclo sugli elementi che non sono attraversati da Γ e in seguito un altro ciclo sugli elementi tagliati.
Per gli elementi non tagliati la costruzione della matrice locale è analoga al caso FEM standard [20]. Vi è però una differenza molto importante rispetto al metodo **addDiffusion()** presente nella classe **ADRAssembler** riguardante l'assemblaggio. Difatti, in **ADRAssembler** la matrice locale è assemblata a partire dell'ID locale dell'elemento, grazie al quale è possibile risalire ai gradi di libertà globali. Per via delle modifiche fatte alla gestione dei gradi di libertà presentate nella sezione 3.3, l'assemblaggio in **XfemAssembler** deve essere fatto a partire dall'ID locale virtuale dell'elemento, attraverso cui è possibile risalire ai gradi di libertà.
Per i $K \in \mathcal{G}_h$, invece, il calcolo delle matrici locali è più complesso e viene presentato nello pseudo-codice 6.

Algoritmo 6 Costruzione della matrice di *stiffness* per elementi tagliati

Estrai gli elementi tagliati;
for $iE = 0 \to nE$ **do**
  {ciclo su tutti gli elementi tagliati}
  for $ise = 0 \to nse$ **do**
    {ciclo sui sotto-elementi}
    Calcolo dei nodi di quadratura P_q, dei pesi w_q e dello jacobiano della trasformazione del sotto-elemento ise;
    Valuta il valore dei gradienti delle funzioni di base $\nabla\phi_i$ dell'elemento iE nei nodi di quadratura P_q;
    Costruzione della matrice di *stiffness* locale a partire dai valori $\nabla\phi_i(P_q)$;
    {si integrano le funzioni di base dell'elemento sul sotto-elemento}
    Assemblaggio della matrice locale a partire dall'ID virtuale locale dell'elemento iE associato alla regione del sotto-elemento ise;
  end for
end for

Da un punto di vista implementativo, l'integrazione sul sotto-elemento del prodotto dei gradienti delle funzioni di base dell'elemento tagliato K è fatto costruendo due **CurrentFE**: uno per il sotto elemento e l'altro per l'elemento tagliato. Il secondo permette di calcolare i nodi di quadratura corretti, i pesi e la trasformazione jacobiana, mentre il primo di questi permette il calcolo dei valori dei gradienti delle funzioni di base nei nodi di quadratura del

sotto-elemento[7].
E' importante notare che l'implementazione fornita permette di gestire elementi attraversati anche da più di un *level set* dal momento che l'integrale è effettuato su ogni regione in cui l'elemento deve essere rappresentato e ogni contributo viene assemblato in modo coerente alle regioni.

Termini di salto attraverso l'interfaccia Consideriamo gli altri tre termini della 3.1, nel caso di un solo *level set* intersecante per ragioni di semplicità[8]. Presenteremo un'analisi dettagliata per il solo termine di penalità dal momento che la filosofia dietro l'implementazione degli altri due è la medesima.
Iniziamo quindi a sviluppare il termine di penalità:

$$\begin{aligned}\lambda([U],[\phi])_\Gamma &= \lambda(U_1 - U_2, \phi_1 - \phi_2)_\Gamma \\ &= \lambda(U_1,\phi_1)_\Gamma + \lambda(U_1,\phi_2)_\Gamma - \lambda(U_2,\phi_1)_\Gamma - \lambda(U_2,\phi_2)_\Gamma\end{aligned}$$

Considerando che le funzioni di base dello spazio di approssimazione sono lineari e valgono 1 nel nodo corrispondente e 0 negli altri,abbiamo che:

$$\begin{aligned}\lambda(U_1,\phi_1)_\Gamma &= \lambda(\sum_j U_1^j \phi_j, \sum_i \phi_i)_\Gamma \\ &= \lambda \sum_{K\in\mathcal{G}_\langle} (\sum_{j\in K} U_1^j \phi_j, \sum_{i\in K} \phi_i)_{\Gamma_K}\end{aligned}$$

Dato quindi un elemento $K \in \mathcal{G}_h$ si devono costruire le seguenti matrici locali (il pedice, in questo caso, indica il nodo locale in cui la funzione di base vale 1) :

$$M = \begin{bmatrix} \int_{\Gamma_K} \phi_1'\phi_1' d\gamma & \int_{\Gamma_K} \phi_1'\phi_2' d\gamma & \cdots & \int_{\Gamma_K} \phi_1'\phi_4' d\gamma \\ \int_{\Gamma_K} \phi_2'\phi_1' d\gamma & \int_{\Gamma_K} \phi_2'\phi_2' d\gamma & \cdots & \int_{\Gamma_K} \phi_2'\phi_4' d\gamma \\ \vdots & \vdots & \ddots & \vdots \\ \int_{\Gamma_K} \phi_4'\phi_1' d\gamma & \int_{\Gamma_K} \phi_4'\phi_2' d\gamma & \cdots & \int_{\Gamma_K} \phi_4'\phi_4' d\gamma \end{bmatrix}$$

$$P = \begin{bmatrix} \int_{\Gamma_K} \phi_1'\phi_1'' d\gamma & \int_{\Gamma_K} \phi_1'\phi_2'' d\gamma & \cdots & \int_{\Gamma_K} \phi_1'\phi_4'' d\gamma \\ \int_{\Gamma_K} \phi_2'\phi_1'' d\gamma & \int_{\Gamma_K} \phi_2'\phi_2'' d\gamma & \cdots & \int_{\Gamma_K} \phi_2'\phi_4'' d\gamma \\ \vdots & \vdots & \ddots & \vdots \\ \int_{\Gamma_K} \phi_4'\phi_1'' d\gamma & \int_{\Gamma_K} \phi_4'\phi_2'' d\gamma & \cdots & \int_{\Gamma_K} \phi_4'\phi_4'' d\gamma \end{bmatrix}$$

$$N = \begin{bmatrix} \int_{\Gamma_K} \phi_1''\phi_1'' d\gamma & \int_{\Gamma_K} \phi_1''\phi_2'' d\gamma & \cdots & \int_{\Gamma_K} \phi_1''\phi_4'' d\gamma \\ \int_{\Gamma_K} \phi_2''\phi_1'' d\gamma & \int_{\Gamma_K} \phi_2''\phi_2'' d\gamma & \cdots & \int_{\Gamma_K} \phi_2''\phi_4'' d\gamma \\ \vdots & \vdots & \ddots & \vdots \\ \int_{\Gamma_K} \phi_4''\phi_1'' d\gamma & \int_{\Gamma_K} \phi_4''\phi_2'' d\gamma & \cdots & \int_{\Gamma_K} \phi_4''\phi_4'' d\gamma \end{bmatrix}$$

[7] Si osservino le routine **computePhiOnImposedNodes** e **computeDphiOnImposedNodes** aggiunti alla classe **CurrentFE**. Questi metodi, dato un **CurrentFE** calcolano i valori di ϕ e $\nabla\phi$ nei nodi di quadratura dell'elemento passato come parametro in input. Per fare ciò viene chiamata la routine **coordBackMap** che permette, dato un punto generico in un elemento, di risalire alle sue coordinate sull'elemento di riferimento.

[8] L'implementazione gestisce in ogni caso anche il caso di taglio doppio.

La matrice λM sarà assemblata coerentemente con i gradi di libertà di K', λN con quelli di K'' e infine la matrice $-\lambda P$ e la sua trasposta saranno assemblate opportunamente in modo da accoppiare il problema tra le due parti dell'elemento.
Poiché $\phi'_j = \phi''_j$, le matrici M,N e P sono in realtà la stessa matrice, cioè che cambia è la posizione in cui esse vengono assemblate.
Nello pseudo-codice 7 viene mostrato in dettaglio come opera il metodo **addPenalty()**.

Algoritmo 7 Costruzione della matrice di penalità

Estrai gli elementi tagliati;
for $iE = 0 \to nE$ **do**
  {ciclo su tutti gli elementi tagliati}
  for $ir = 0 \to nr$ **do**
    {ciclo sulle regioni associate alle facce che tagliano l'elemento}
    $vEleRegion1 \leftarrow$ *flag* della regione adiacente al *level set* per cui vale $f(\mathbf{x}) < 0$;
    $vEleRegion2 \leftarrow$ *flag* della regione adiacente al *level set* per cui vale $f(\mathbf{x}) > 0$;
    $volK1 \leftarrow$ volume totale dei sotto-elementi nella regione $vEleRegion1$;
    $volK2 \leftarrow$ volume totale dei sotto-elementi nella regione $vEleRegion2$;
    for $if = 0 \to nf$ **do**
      {ciclo sulle facce nella regione ir che tagliano l'elemento}
      Controllo sulla coerenza della direzione della normale alla faccia if;
      Costruzione della matrice di locale;
      Assemblaggio della matrice locale a partire dagli ID virtuali locali dell'elemento iE associato alle regioni $vEleRegion1$ e $vEleRegion2$;
    end for
  end for
end for

Anche nel caso della matrice locale di penalità la difficoltà risiede nell'integrazione del prodotto della traccia delle funzioni di base per le loro derivate nella direzione normale sulla faccia di taglio. Si utilizzano dunque i nodi di quadratura della faccia, ma i valori della funzione nei nodi devono essere calcolati a partire dalla funzioni di base dell'elemento tagliato K. In modo analogo a quanto fatto per lo *stiffness* si sfrutterà un oggetto di classe **CurrentBoundaryFE**, grazie a cui sono calcolati nodi di quadratura e trasformazione jacobiana, e un oggetto di classe **CurrentFE**, costrutito a partire dall'elemento tagliato, per la valutazione delle ϕ e della loro derivata normale.
Analogamente a quanto fatto per la penalità, sviluppiamo il termine:

$$\begin{aligned}(\{\alpha \nabla_{\mathbf{n}} U\}, [\phi])_\Gamma = &(\alpha_1 \kappa_1 \nabla_{\mathbf{n}} U_1, \phi_1)_\Gamma - (\alpha_1 \kappa_1 \nabla_{\mathbf{n}} U_1, \phi_2)_\Gamma \\ &+ (\alpha_2 \kappa_2 \nabla_{\mathbf{n}} U_2, \phi_1)_\Gamma - (\alpha_2 \kappa_2 \nabla_{\mathbf{n}} U_2, \phi_2)_\Gamma\end{aligned}$$

Per la costruzione di questi termini è necessario il calcolo della seguente matrice locale L:

$$L = \begin{bmatrix} \int_{\Gamma_K} \nabla\phi_1\phi_1 d\gamma & \int_{\Gamma_K} \nabla\phi_1\phi_2 d\gamma & \cdots & \int_{\Gamma_K} \nabla\phi_1\phi_4 d\gamma \\ \int_{\Gamma_K} \nabla\phi_2\phi_1 d\gamma & \int_{\Gamma_K} \nabla\phi_2\phi_2 d\gamma & \cdots & \int_{\Gamma_K} \nabla\phi_2\phi_4 d\gamma \\ \vdots & \vdots & \ddots & \vdots \\ \int_{\Gamma_K} \nabla\phi_4\phi_1 d\gamma & \int_{\Gamma_K} \nabla\phi_4\phi_2 d\gamma & \cdots & \int_{\Gamma_K} \nabla\phi_4\phi_4 d\gamma \end{bmatrix}$$

La routine che costruisce queste matrici locali e le assembla è molto simile a quella che calcola il termine di penalità, per cui non riportiamo ulteriori dettagli. L'assemblaggio in questo caso merita particolare attenzione per via del fatto che la matrice non è più simmetrica e per via dei coefficienti che la moltiplicano a seconda del termine che si sta assemblando. A titolo di esempio, la matrice $\alpha_1\kappa_1 L$ sarà assemblata coerentemente con i gradi di libertà dell'elemento K', mentre per assemblare $-\alpha_2\kappa_2 L$ bisognerà conoscere i gradi di libertà di K''. Discorso analogo per gli altri due termini.
Il termine restante al primo membro è in realtà il trasposto di quello appena descritto ed è inserito per simmetrizzare il sistema lineare.

Secondo membro La costruzione dei termini a secondo membro ricalca i passaggi mostrati sinora per la costruzione delle matrici. La difficoltà risiede ancora una volta nell'integrazione delle funzioni di base di un elemento $K \in \mathcal{G}_h$ su parti di esso o su facce di taglio e nell'assemblaggio. L'implementazione è contenuta nel file XfemAssembler.hpp.

3.4.3 Imposizione delle condizioni al bordo

Al momento è stata implementata solamente la possibilità di imporre della condizioni essenziali sulla frontiera del dominio $\tilde{\Omega}$. E' quindi una delle parti maggiormente da sviluppare in questa implementazione del metodo [14] in `LifeV`.
Le classi che gestiscono le condizioni al bordo sono:

- **BCFunctionBase**, che definisce una funzione da interpolare opportunamente sui nodi sulla frontiera;
- **BCHandler**, classe attraverso cui si definiscono le condizioni al bordo su ogni parte di frontiera[9];

In BCManage.hpp sono inoltre contenute le routine che, data la matrice globale del problema ed il vettore a secondo membro, impongono le condizioni al contorno nel sistema.
Per imporre le condizioni di Dirichlet su alcune regioni della frontiera è stata implementata la routine **bcUpdateXfem()**. Essa impone le condizioni al bordo su tutti i gradi di libertà relativi ad un entità geometrica di bordo, attraverso un doppio ciclo su elementi di bordo e regioni associate agli elementi. L'implementazione è contenuta nel file BCHandler.hpp.

3.5 Visualizzazione dei risultati

Dopo aver risolto il sistema lineare associato al problema discretizzato 1.7 rimane il problema di come visualizzare una soluzione discontinua all'interno di un elemento. L'idea è quella di utilizzare la ritriangolazione degli elementi attraversati da Γ per costruire una griglia *matching*. A questo punto bisogna interpolare la soluzione ottenuta sullo spazio ad elementi finiti arricchito (e quindi sulla *mesh* originale) su uno spazio discontinuo, almeno sui nodi che si trovano su Γ, costruito sulla nuova *mesh*.

[9] In genere, ogni entità della *mesh* in `LifeV` possiede una *flag* nel *marker* di base che indica a quale parte del dominio appartiene. Tramite questa *flag* è possibile differenziare parti di frontiera per un entità che si trova sul bordo.

3.5.1 Esportazione della *mesh* tagliata

Si vuole sostituire la *mesh* partizionata su cui sono stati fatti i calcoli con una nuova in cui gli elementi tagliati vengono sostituiti dai sotto-elementi contenuti nei *marker*.
Si procede dunque nel seguente modo. Data la *mesh* partizionata, si inseriscono inizialmente i punti di intersezione (relativi alla sola partizione considerata) e in seguito, ciclando sugli elementi, si sostituisce l' elemento tagliato con il primo *sub-element* contenuto nel *marker* mentre gli altri vengono inseriti in fondo al vettore degli elementi.
E' importante far notare che negli elementi non tagliati è necessario ripristinare i puntatori ai vertici dal momento che l'aggiunta di punti comporta la riallocazione della variabile che li contiene.
Al termine di questa operazione di inserimento vengono puliti i vecchi *marker* degli elementi[10] e vengono svuotati i vettori di lati e facce della *mesh*. Questi vengono poi ricostruiti dalla routine **checkMesh3D**, che oltre a ricalcolare lati e facce compie una serie di controlli volti a garantire la correttezza della *mesh* da un punto di vista geometrico e di convenzioni necessarie alle funzionalità di `LifeV` come, ad esempio, la numerazione degli ID.
Vengono inoltre pulite le variabili **M_mapCutEle**, **M_pointsOnEdges** e **M_pointsOnFaces** in **XfemMeshHandler** facendo attenzione a riallocare la memoria.
La modifica della *mesh* partizionata piuttosto che di quella non ancora partizionata comporta i seguenti vantaggi:

- maggiore velocità, in quanto ogni processore modifica solamente la propria partizione e non tutta la *mesh* (dal momento che questa operazione è la parte più lenta, il fatto di essere svolta in parallelo migliora sensibilmente la velocità di esecuzione, come verrà mostrato nei test);
- evita problemi dovuti a griglie non conformi prodotte dalla triangolazione degli elementi tagliati;

D'altro canto questa scelta necessita la conoscenza a priori degli ID globali dei nuovi elementi aggiunti[11], altrimenti resa difficile dal fatto che per avere una variabile globale che si incrementi ogni volta che un elemento viene aggiunto in una partizione sarebbe necessario chiedere ad ogni processo di informare gli altri dell'avvenuta aggiunta.
La costruzione di questa nuova *mesh* è contenuta nella routine **modifiedMesh()** nella classe **XfemMeshHandler**.

3.5.2 Interpolazione della soluzione sulla *mesh* tagliata

Al fine di visualizzare le soluzione e di calcolare la norma L_2 dell'errore è necessario interpolare la soluzione ottenuta su uno spazio ad elementi finiti costruito sulla *mesh* tagliata.

La classe XfemInterpolation Notiamo innanzitutto che durante l'esecuzione del metodo **modifiedMesh()** della classe **XfemMeshHandler** le informazioni circa gli elementi tagliati

[10] Il metodo **clearMarker()** resetta il peso dell'elemento e cancella le mappe contenenti le *sub entities*, mentre **clearIntersectionPoints()** pulisce la memoria occupata dalla variabile **S_intersectionPoints**.

[11] Il fatto che il taglio della *mesh* sia un'operazione compiuta in seriale permette di avere facilmente queste informazioni.

nella *mesh* originale vengono perse. L'interpolazione della soluzione sulla ritriangolazione di un elemento tagliato necessita tuttavia di conoscere le coordinate dei vertici e la mappa dei gradi di libertà dell'elemento originale, persi appunto nel momento in cui l'elemento è sostituito dal primo sotto-elemento contenuto nel *marker*.
Una soluzione molto impegnativa dal punto di vista della memoria occupata è quella di conservare la *mesh* originale e poi interpolare la soluzione essendo note tutte le informazioni necessarie. Un 'altra possibilità è quella di conservare solamente le informazioni strettamente necessarie in una classe che conterrà anche le routine di interpolazione.
E' stata quindi implementata la classe **XfemInterpolation** la cui definizione è contenuta nel file XfemInterpolation.hpp. Riportiamo la dichiarazione di questa classe:

```
class XfemInterpolation
{

public:
XfemInterpolation( );

template< typename MeshType, typename MapType >
void initialize( boost::shared_ptr< FESpace<MeshType, MapType> > feSpace );

template< typename vector_type, typename MeshType, typename MapType >
void interpolateSolution( const vector_type& vector, vector_type& interpolatedSol,
boost::shared_ptr< FESpace<MeshType, MapType> > feSpace );
void showMe( std::ostream & out = std::cout ) const;

private:

const ReferenceFE* M_originalRefFEtype;
const ReferenceFE* M_cutMeshRefFEtype;

boost::shared_ptr<DOFXfem> M_originalDof;

std::map<UInt,UInt> M_cutEleToEle;

std::multimap< UInt, UInt > M_eleToPoint;
};
```

Questa classe contiene dunque due puntatori a **ReferenceFE**. Uno è per l'elemento di riferimento dello spazio ad elementi finiti sulla *mesh* originale, l'altro per quello dello spazio costruito sulla *mesh* tagliata. In genere essi sono uguali.
Tra gli attributi è presente inoltre uno *smart pointer* ad un oggetto di classe **DOF**, e due mappe. Dalla prima mappa, **M_cutEleToEle**, dato l'ID globale di un elemento sulla *mesh* tagliata si ottiene l'ID locale dell'elemento intero nella *mesh* originale[12]. Dalla seconda mappa,

[12]Questa mappa riguarda in realtà solo gli elementi ritriangolati e sostituiti dalla loro ritriangolazione. Tramite questa mappa infatti è possibile risalire all'elemento originale. Se un elemento non è tagliato, esso non subisce alcuna modifica: ID locale, globale coordinate e *flag* non vengono modificati nel corso dell'esecuzione della

dato l'ID locale di un elemento della *mesh* originale si ricavano gli ID dei suoi vertici.
Il metodo **initialize()** deve essere eseguito prima di chiamare il metodo **modifiedMesh()** passando come parametro un puntatore al **FESpace** rispetto a cui è stata calcolato il vettore soluzione. Questo metodo costruisce le due mappe presenti e assegna alle variabili **M_originalDof** e **M_originalRefFEtype** rispettivamente il **DOF** e il **ReferenceFE** contenuti nel **FESpace** dato come parametro in input.
Dopo l'esecuzione di questa routine sono state archiviate le informazoni necessarie per l'interpolazione e si può procedere alla creazione della *mesh* tagliata e alla definizione di un nuovo **FESpace**.
Per interpolare la soluzione sul nuovo spazio,bisogna chiamare la routine **interpolateSolution()** che riceve in ingresso due vettori e un puntatore al nuovo **FESpace**. Uno dei due vettori è passato come *const* ed è il vettore soluzione precedentemente calcolato, l'altro è il vettore in cui dovrà essere inserita la soluzione interpolata.
Il metodo **showMe()** è stato implementato per visualizzare le mappe contenute nella classe in modo da facilitare un eventuale *debug* del programma.
Al momento, questa classe è capace di interpolare una soluzione, calcolata su uno spazio arricchito, su uno spazio discontinuo cioè costruito a partire da una gestione discontinua dei gradi di libertà.

La classe DiscountinuousDOF Quanto presentato sinora permette di avere una griglia *matching* rispetto a Γ, e di visualizzare la soluzione su una ritriangolazione degli elementi originali tagliati. Questo non è però sufficiente perché la soluzione sia discontinua attraverso Γ. L'assegnazione dei gradi di libertà contenuta in **DOF** non consente discontinuità: elementi adiacenti hanno gli stessi gradi di libertà sulla faccia in comune.
E' stata allora implementata la classe **DiscontinuousDOF** che assegna i gradi di libertà a partire dall'ID globale dell'elemento e non a partire dall'ID globale dell'entità geometrica come illustrato nello pseudo-codice 8.

In questo modo ogni elemento possiede gradi di libertà distinti dagli altri ed è possibile rappresentare una soluzione discontinua.
La classe **DiscountinuousDOF** eredita pubblicamente da **DOF**. L'assegnazione dei gradi di libertà viene eseguita dalla routine **updateDiscontinuousDOF()**, che viene chiamata al momento dell'istanziazione di un **FESpace** se la *flag* **typeDOF** del costruttore di quest'ultimo ha valore 2.
Questa presentata non è la soluzione più efficiente in quanto il numero dei gradi di libertà totali è dato dal numero degli elementi per il numero di gradi di libertà locali. Il vettore soluzione costruito su questo spazio sarà dunque molto più lungo di un eventuale vettore soluzione in cui i gradi di libertà sono assegnati in modo discontinuo solo lungo Γ. Tuttavia, l'implementazione di questa classe, che assegna i gradi di libertà in modo da avere un spazio discontinuo di elementi finiti, è più generale e può essere utile in `LifeV` nel caso si vogliano implementare tecniche numeriche che facciano uso di questo tipo di spazi, ad esempio i *Discontinuous Galerkin*.

La classe ExporterHDF5FromDOF L'ultimo passo necessario per l'esportazione di una soluzione in modo che sia visualizzabile in `ParaView` è quello di fare alcune modifiche alla classe

routine **modifiedMesh()**.

Algoritmo 8 Assegnazione dei gradi di libertà sui vertici degli elementi in uno spazio discontinuo

$gCount \leftarrow 0$;
if gradi di libertà associati ai vertici > 0 **then**
  for $ie = 0 \rightarrow ne$ **do**
    {ciclo sugli elementi della partizione}
    $lc \leftarrow 0$;
    for $iv = 0 \rightarrow nv$ **do**
      {ciclo sui vertici dell'elemento}
      for $i = 0 \rightarrow nd$ **do**
        {nd è il numero di gradi di libertà per ogni vertice}
        Assegna all'lc-esimo grado di libertà locale dell'elemento ie il grado di libertà globale $gCount + ie * nd + lc$;
        $lc \leftarrow lc + 1$;
      end for
    end for
  end for
end if

ExporterHDF5. Per evitare conflitti con la *testsuite* di `LifeV` la classe **ExporterHDF5** è stata duplicata. All'interno della sua copia, **ExporterHDF5FromDOF**, sono state effettuate le modifiche che permettono di esportare una soluzione costruita su uno spazio ad elementi finiti discontinui. In vista di un inserimento di questa classe nel *master* di `LifeV`, sarà necessario renderla più generale e possibilmente definirla come una classe figlia di **ExporterHDF5**.

Capitolo 4

Risultati

4.1 Verifica dei risultati teorici

Al fine di validare il codice implementato, sono stati predisposti alcuni test volti a verificare alcuni risultati teorici relativi al metodo in [14]. E' stato verificato l'ordine di convergenza del metodo secondo la norma L_2 ed è stato fatto un confronto con un metodo FEM *unfitted*. Si è poi analizzato il comportamento dell'autovalore massimo e minimo della matrice globale al variare della posizione del *level set*.

Il test XFEM in `LifeV` Per effettuare i test che seguono è stato aggiunto un test alla *testsuite* di `LifeV`. Nel file main.cpp nella cartella test_XFEM si può trovare il codice del test. Grazie ad esso si comprende come utilizzare le strutture dati che sono state sin qui presentate, quali routine chiamare e con quale ordine.

Il caso test Per la verifica dei risultati teorici risolviamo in un dominio 3D il problema test usato in [14].
Consideriamo la soluzione dell'equazione differenziale ordinaria:

$$-\sum_i \frac{d}{dz}\left(\alpha_i \frac{du_i}{dz}\right) = 1$$

avendo imposto le condizioni:

$$\begin{aligned}
&[u(1/2)] = 0 \\
&\alpha_1 \frac{du_1}{dz}(1/2) = \alpha_2 \frac{du_2}{dz}(1/2) \\
&u_1(0) = 0 \\
&u_2(0) = 0
\end{aligned}$$

E' un problema monodimensionale sul dominio $\tilde{\Omega} = (0,1)$ con un'interfaccia in $z = 1/2$.
Si risolverà numericamente questa equazione sul dominio tridimensionale $\tilde{\Omega} = (0,1) \times (0,1) \times (0,1)$ con condizioni di Newmann omogenee in $x = 0$, $x = 1$, $y = 0$ e $y = 1$. Il salto sulla

derivata conormale è invece imposto sul piano $z = 1/2$.
E' nota la soluzione analitica di questo problema, data da:

$$u_1(z) = \frac{(3\alpha_1 + \alpha_2)z}{4\alpha_1^2 + 4\alpha_1\alpha_2} - \frac{z^2}{2\alpha_1}$$
$$u_2(z) = \frac{\alpha_2 - \alpha_1 + (3\alpha_1 + \alpha_2)z}{4\alpha_2^2 + 4\alpha_1\alpha_2} - \frac{z^2}{2\alpha_2}$$

Nei test che verranno mostrati si è posto $\alpha_1 = 1/2$ e $\alpha_2 = 20$.
Per la risoluzione del sistema lineare si è fatto uso dei pacchetti di `Trilinos` [16], in particolare del solutore `AztecOO` con precondizionatore `IfPack` [21].

4.1.1 Ordine di convergenza

Vogliamo verificare che l'ordine di convergenza in norma L_2 del metodo implementato è quadratico rispetto ad h come mostra la stima (1.22). Si è anche voluto fare un confronto con un metodo FEM *unfitted*.
Il problema test è stato quindi risolto sulla medesima *mesh*, utilizzando i due metodi. In tabella 4.1 mostriamo i valori della norma L_2 dell'errore per i due metodi su griglie via via più fini.

Tabella 4.1: Confronto tra le norme L_2 dell'errore per XFEM e *unfitted* FEM

h	XFEM	FEM
0.15746	0.000802	0.007898
0.10188	0.000303	0.005134
0.08242	0.000189	0.004162
0.05587	8.997e-5	0.002824
0.04224	6.297e-5	0.002137

Se costruiamo un grafico del logaritmo dell'errore in funzione di h otteniamo la figura 4.2

Possiamo affermare che il risultato è soddisfacente. Osserviamo infatti che la norma dell'errore calcolata con il metodo XFEM converge quadraticamente rispetto ad h. Il metodo implementato inoltre è sensibilmente più accurato del metodo FEM, fatto che giustifica la necessità di sviluppare metodi più performanti qualora la soluzione presenti salti o irregolarità.

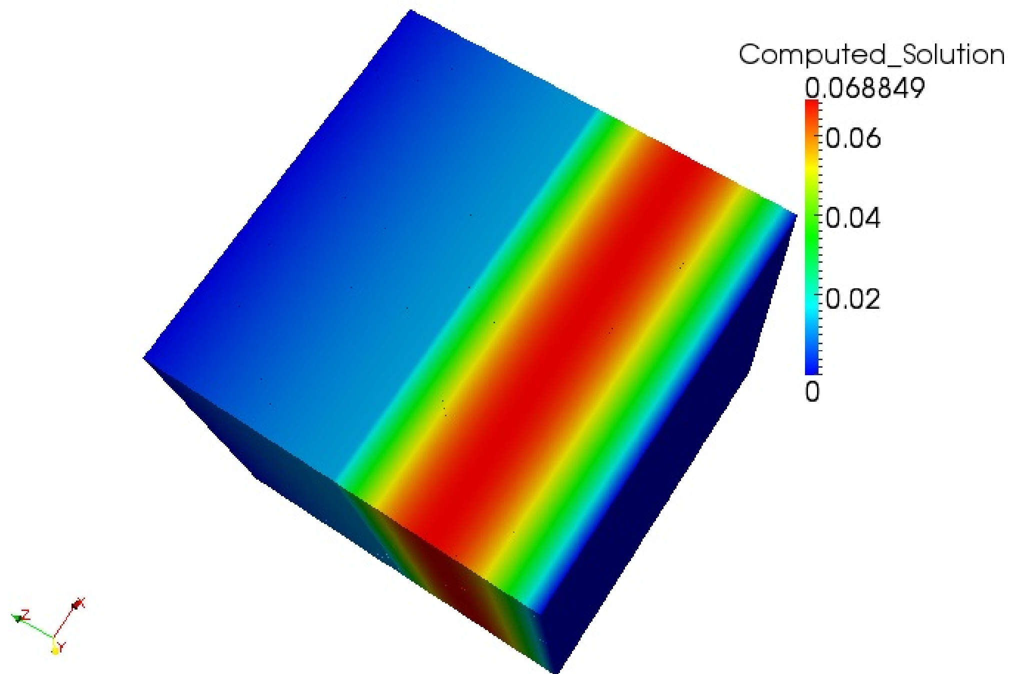
4.1.2 Analisi degli autovalori

Sono stati verificati anche alcuni risultati teorici sulle stime degli autovalori (massimo e minimo) della matrice globale (il cui *pattern* di sparsità è mostrato in figura 4.3).
Riportiamo brevemente i risultati teorici, per i dettagli si veda [5].
Per l'autovalore massimo vale la stima:

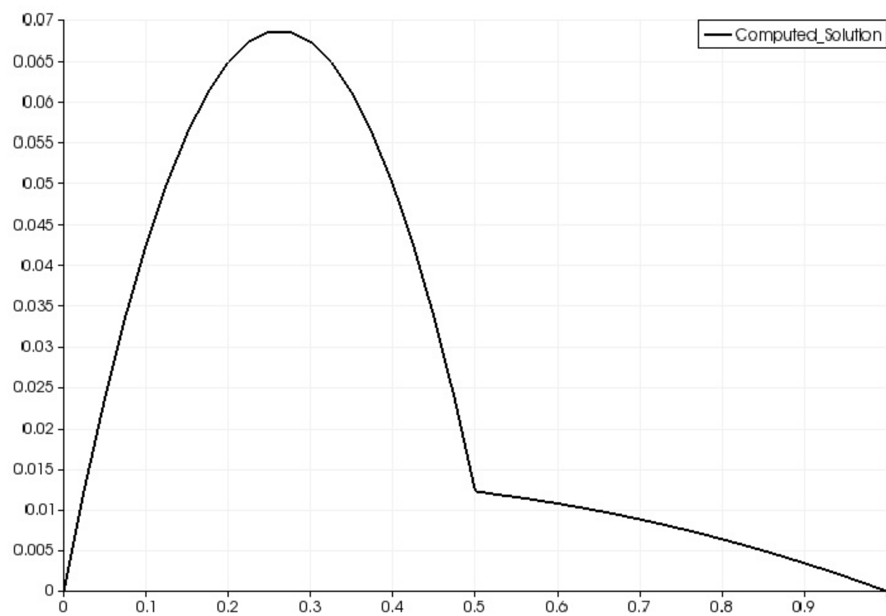
$$\lambda_{max} \leq \tilde{M} h^d (1 + \tilde{C} h^{-2}) \tag{4.1}$$

dove le costanti non dipendono dalla posizione dell'interfaccia Γ e d è la dimensione del dominio.
Per l'autovalore minimo vale invece:

$$\lambda_{min} \geq \bar{\alpha}_C C_{min}(h, \kappa_i) \tag{4.2}$$



(a) Soluzione sul dominio 3D



(b) Soluzione lungo una retta parallela all'asse z

Figura 4.1: Soluzione approssimata del caso test

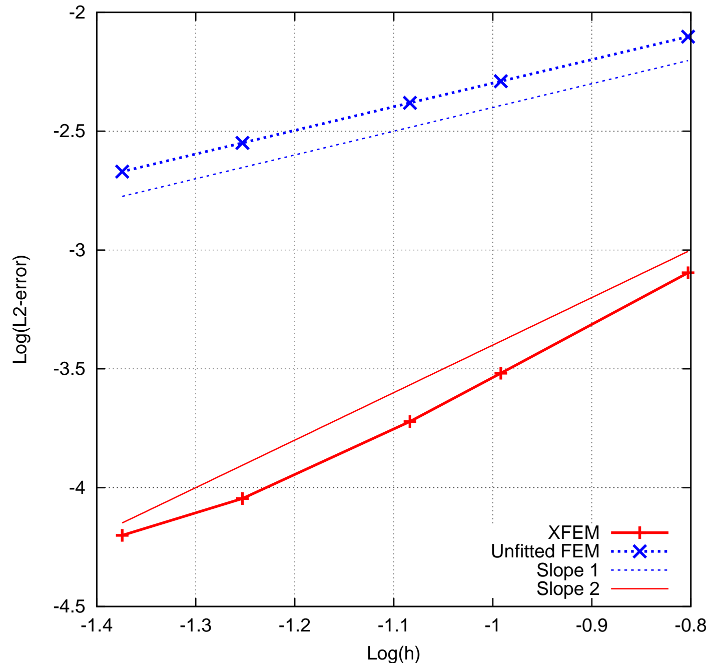


Figura 4.2: Andamento della norma L_2 dell'errore per metodo XFEM e *unfitted* FEM

dove per la definizione delle costanti si faccia riferimento a [5]. In quest'ultima è importante notare come l'autovalore minimo varia in funzione del rapporto tra i volumi delle parti in cui un elemento viene tagliato e quindi varia in funzione della posizione del *level set*.

Mostriamo ora i risultati ottenuti al variare della posizione del *level set* e al variare di h.

In funzione del rapporto tra i volumi K_i Risolviamo il problema discreto spostando di volta in volta il *level set* in modo da avvicinarlo ad un piano su cui giacciono le facce di alcuni elementi della *mesh*. In questo modo il minimo rapporto tra K_1 e K_2 per gli elementi $K \in \mathcal{G}_h$ tende a diminuire. In tabella 4.2 mostriamo i risultati ottenuti. Osserviamo che l'autovalore

Tabella 4.2: Autovalore massimo e minimo al variare della posizione del *level set*

$\min(K_1/K_2)$	λ_{max}	λ_{min}
0.125	3.481	0.00806
0.015625	3.468	0.00812
0.001	3.465	8.44e-4
1.25e-4	3.464	1.02e-4
1.56e-5	3.460	1.26e-5
1.25e-7	3.463	1.0029-7

massimo rimane pressoché costante mentre l'autovalore minimo diminuisce linearmente con il minimo rapporto tra i volume delle due parti dell'elemento. Questi risultati sono in linea con la teoria.

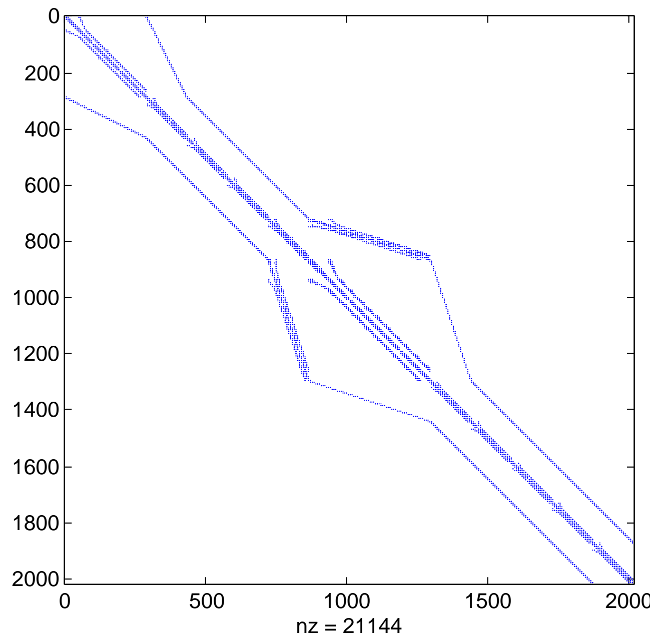


Figura 4.3: *Pattern* di sparsità della matrice del test

Variazione di h In questo caso, il problema discreto è stato risolto con una griglia via via più fine. In tabella 4.3 sono indicati i valori degli autovalori calcolati in funzione del passo di griglia h. Osserviamo che sia l'autovalore massimo sia quello minimo sono influenzati dal

Tabella 4.3: Autovalore massimo e minimo al variare di $h = 1/N$

N	λ_{max}	λ_{min}	$\lambda_{max}/\lambda_{min}$
5	4.40769	0.02148	2.05e+2
11	1.1505	0.00427	2.67e+2
19	1.00001	0.00106	9.51e+2
29	1	3.29e-4	3.03e+3
39	1	1.42e-4	7.002e+3

variare di h. L'autovalore massimo tende a diminuire stabilizzandosi intorno al valore 1, mentre quello minimo decresce in modo pressoché quadratico con h.
Il rapporto $\lambda_{max}/\lambda_{min}$ cresce approssimativamente come h^{-2} e questo risultato è in linea con la teoria che prevede:

$$\frac{\lambda_{max}}{\lambda_{min}} \leq \frac{\tilde{M}h^d(1+\tilde{C}h^{-2})}{\bar{\alpha_C}C_{min}(h,\kappa_i)} \approx h^{-2}$$

Con questi test abbiamo dunque analizzato il variare dell'ampiezza dello spettro delle matrice globale. E' stato verificato che la posizione del *level set* influenza solamente l'autovalore minimo, mentre il passo h della *mesh* agisce su entrambi gli autovalori.

4.2 Performace del codice

Altri test sono stati implementati per valutare il codice implementato da un punto di vista delle performance di calcolo. In particolare sono stati analizzati la scalabilità e il *load balancing.*

4.2.1 Scalabilità

L'implementazione fatta ha posto particolare attenzione a garantire la possibilità di eseguire calcoli in parallelo, nel rispetto della direzione presa da qualche anno da `LifeV`.
Il codice è stato implementato per fare gran parte dei calcoli in parallelo. Solo il calcolo delle intersezioni e delle ritriangolazioni degli elementi tagliati è svolto in seriale da ciascun processore e così pure la definizione degli ID globali virtuali delle entità su cui sono presenti gradi di libertà[1]. Dopo queste due operazioni è possibile partizionare la *mesh* e il calcolo delle matrici, risoluzione del sistema lineare e esportazione della soluzione sono svolti in parallelo.
In tabella 4.4 mostriamo i risultati ottenuti circa la scalabilità del codice implementato.

Tabella 4.4: Tempi di esecuzione del calcolo su n processori

n	Tempo totale	Esclusa esportazione
1	12693.5	846.5
2	2305.6	230.7
4	975.3	53.43
8	426.9	22.29
16	158	11.21
32	83.4	9.74
64	60.4	8.35

Innanzitutto osserviamo che in tabella sono mostrati il tempo di esecuzione totale e il tempo di esecuzione esclusa la fase di esportazione della soluzione. Come si vede in tabella, la differenza è sostanziale in quanto la fase di esportazione della soluzione comprende la fase di creazione della *mesh* tagliata (si veda 3.5) che non è ancora ottimizzata. Difatti l'aggiunta di punti ed elementi nuovi comporta la riallocazione della memoria occupata dalla lista di punti e volumi. Per migliorare questa fase è necessario stimare a priori il numero di punti da aggiungere alla partizione che si sta modificando[2] in modo da riallocare una sola volta la memoria della variabile.
Questo può essere anche la spiegazione del fatto che nel passare da uno a due processori il tempo di calcolo scala in modo superlineare. Vediamo che il codice scala linearmente tra 2 e 16 processori. Sopra i 16 processori i tempi di esecuzione diminuiscono ma in modo sub-lineare.
Potrebbe essere interessante indagare sulle cause di questo comportamento ma nell'ambito di questa tesi riportiamo solamente il risultato ottenuto, comunque positivo, circa la scalabilità del codice senza indagare ulteriormente sulle cause della scalabilità sublineare a partire da 16 processori.

[1] Queste operazioni necessitano la continua definizione di ID globali come si è visto nei capitoli precedenti. Per ovviare alle difficoltà dovute ad una continua comunicazione tra i processori per aggiornare le variabili globali si è preferito svolgere questa parte del calcolo in seriale.

[2] La stima del numero di elementi da aggiungere è fatta proprio in fase di aggiunta dei punti.

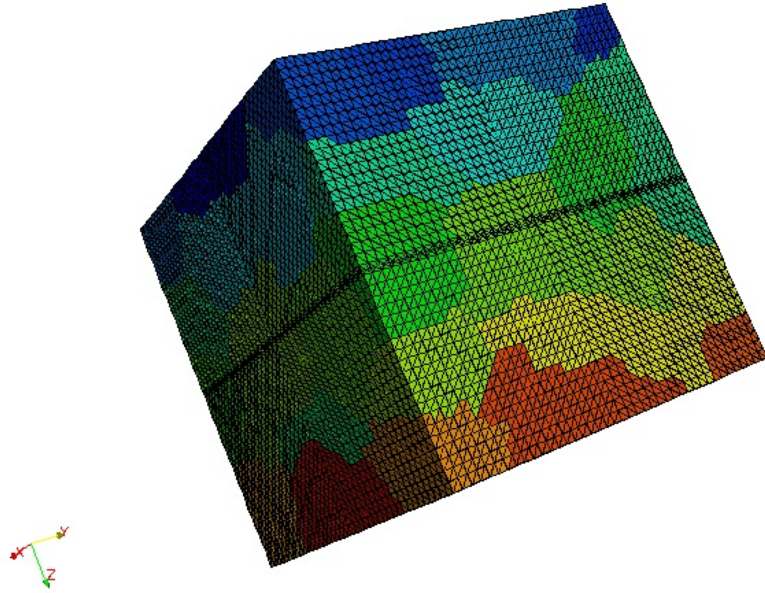


Figura 4.4: Decomposizione del dominio per il calcolo a 64 processori

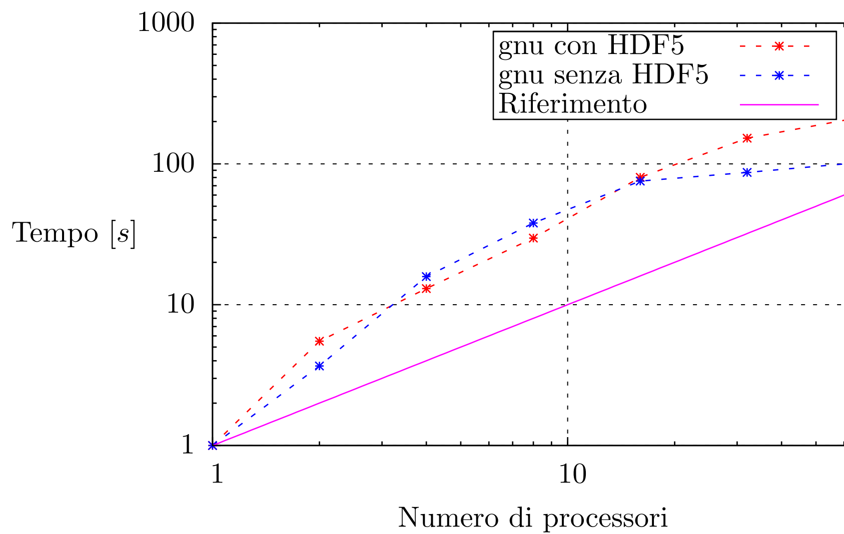


Figura 4.5: Scalabilità del codice implementato.

I calcoli sono stati eseguiti sul calcolatore LAGRANGE del Cilea: 208 nodi, doppio processore Intel Xeon QuadCore 3.166GHz per nodo e 16 GB di RAM per nodo con una capacità totale di 13 TB.

4.2.2 *Load balancing*

Nella sezione 3.2 sono state descritte le modifiche apportate alla classe **MeshPartitioner** in modo da ottenere un partizionamento pesato, così da garantire un carico di calcolo equilibrato

tra i diversi processori. Questo test vuole verificare se l'implementazione del partizionamento pesato ha un effettivo impatto sul bilanciamento del carico sui diversi processori. E' stato risolto una problema differenziale simile a quanto fatto nelle sezioni precedenti su una *mesh* partizionata in modo standard e abbiamo confrontato i tempi di esecuzione con quelli dello stesso problema risolto risolto su una griglia partizionata in modo pesato.
In tabella 4.5 sono mostrati i risultati ottenuti dalle due simulazioni.

Tabella 4.5: Confronto dei tempi di esecuzione del calcolo su due processori tra partizionamento pesato e non pesato

	Pesato		Non pesato	
Processore	1	2	1	2
Lettura *mesh*	0.27	0.27	0.25	0.26
Taglio *mesh*	0.23	0.24	0.24	0.25
Assemblaggio *stiffness*	1.26	1.2	0.88	1.61
Assemblaggio penalità	0.06	0.06	0	0.12
Assemblaggio salto (I membro)	0.28	0.23	0.01	0.53
Assemblaggio forzante	0.77	0.79	0.56	0.96
Assemblaggio salto (II membro)	0.16	0.14	0	0.31
Assemblaggio matrice globale	2.82	2.88	3.79	3.76
Soluzione del sistema	0.69	0.79	0.6	0.7
Creazione *mesh*	49.81	45.1	0.15	128.96

E' evidente che nel caso considerato il partizionamento pesato riesce a bilanciare il carico di calcolo sui due processori. Si nota infatti che nel caso in cui i calcoli sono stati eseguiti sulla griglia partizionata in modo standard, c'è un netto squilibrio tra i tempi di esecuzione dei due processori. Difatti, essendo tutti gli elementi tagliati sul processore 2, il calcolo per la costruzione della matrice e del vettore forzante è sensibilmente più oneroso per questo processore.
La modifica della classe **MeshPartitioner** è stata quindi utile al fine del miglioramento delle performance del codice, in quanto si è verificato che permette un effettivo ribilanciamento del calcolo quando, per via dell'arricchimento locale, alcuni elementi sono rappresentati da un numero maggiore di gradi di libertà e necessitano un trattamento particolare a livello di calcolo della matrice locale.

4.3 Taglio multiplo

In ultimo, si è voluto testare l'implementazione di un taglio doppio in cui il dominio è attraversato da due *level set* che si intersecano tra loro. Sono quindi presenti elementi in cui è necessario calcolare la triangolazione di quattro o tre sottoparti. L'attribuzione dei gradi di libertà e integrazione dovrà essere fatta in modo da distinguere le diverse regioni in cui l'elemento viene diviso.
Per poter verificare l'accuratezza della soluzione numerica si è risolto il problema presentato

nella sezione 4.1 aggiungendo il *level set* dato dall'equazione:

$$\frac{3}{10}x - \frac{1}{5} + z - \frac{1}{2} = 0$$

Su questa interfaccia, che chiameremo Σ, si è imposta la continuità della soluzione. Il coefficiente di diffusione α non presenta salti attraverso questa interfaccia, in questo modo la soluzione analitica è la medesima del problema già analizzato per un solo *level set*. In pratica, l'aggiunta della seconda interfaccia non introduce ulteriori discontinuità nel parametro α ma impone, in ogni caso, di utilizzare le tecniche di integrazione XFEM presentate in [14] per gli elementi intersecati da Σ.

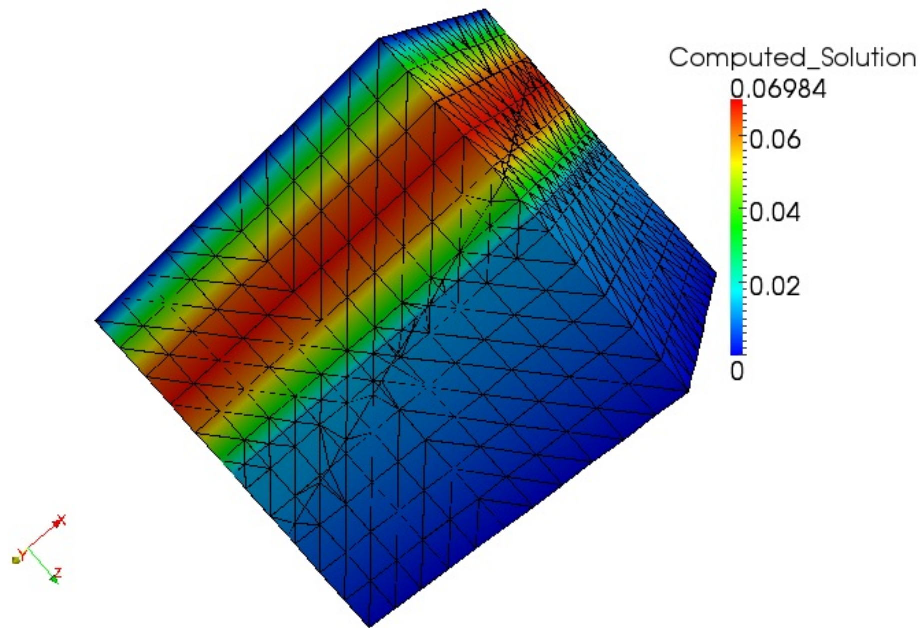


Figura 4.6: Soluzione del test con due *level set* intersecanti il dominio

In figura 4.6 è mostrata la soluzione ottenuta con una *mesh* grossolana ($h = 0.1574$). Si nota la presenza del secondo *level set* attraverso cui la soluzione è continua e non vi è salto sulla derivata conormale.
E' stato fatto anche in questo caso il test di convergenza dell'errore in norma L_2, in modo da verificare se anche in questo caso i risultati numerici sono in linea con la stima[3] (1.22).
I risultati sono mostrati nella tabella 4.6. La decrescita dell'errore al variare di h non è dell'ordine h^2 come nel caso del test con la sola interfaccia Γ. Non sono noti risultati teorici inerenti all'analisi di convergenza del metodo [14] nel caso di due *level set* cui cui confrontare i risultati numerici.
Tuttavia, visto che la strategia utilizzata per la gestione del doppio taglio discende da quanto presentato per una sola interfaccia, ci si sarebbe potuti aspettare dei risultati un pò migliori. Le ragioni di quanto ottenuto possono essere ricercate nel fatto che, quando due *level set* si intersecano, la stima del parametro di penalità λ e il calcolo dei pesi κ_i può essere più delicato.

[3] Si noti che tale stima è ottenuta nel caso in un solo *level set*.

Tabella 4.6: Norma L_2 dell'errore in funzione del passo h per il test con due *level set*

h	Errore
0.15745	0.0007374
0.10188	0.0003196
0.08247	0.0002346
0.05587	0.0001579
0.04224	0.0001268

Ad esempio, è forse necessario penalizzare in modo diverso il salto attraverso le due interfacce, definendo il coefficiente di penalità λ in funzione del salto del parametro α attraverso l'interfaccia.

Conclusioni

L'obiettivo di questa tesi era l'implementazione del metodo XFEM proposto da Hansbo e Hansbo [14] nella libreria di calcolo ad elementi finiti `LifeV` per risolvere un problema di diffusione in un dominio tridimensionale attraversato da una superficie attraverso la quale la soluzione presenta irregolarità per via del salto del coefficiente di diffusione.

Dopo una prima fase di ricerca bibliografica in cui si è approfondito il metodo e le sue proprietà di convergenza e accuratezza, si è passati alla fase implementativa, che ha rappresentato il cuore di questo lavoro. Lo sviluppo del codice ha seguito diverse tappe. Innanzitutto ci si è interessati a sviluppare delle strutture dati che fossero in grado di gestire l'intersezione tra la griglia tridimensionale e l'interfaccia, definita da una funzione *level set*. L'implementazione si è dapprima concentrata sul caso di una sola interfaccia e, una volta verificata la correttezza del codice, è stata sviluppata anche la gestione di un insieme di *level set*.
A questo punto si è trattato di adattare e integrare le strutture dati già presenti in `LifeV` per la risoluzione del problema discreto. La difficoltà maggiore è stata quella incontrata nell'implementazione di classi capaci di gestire in modo adeguato i gradi di libertà. Questo è stato il passaggio chiave della parte del lavoro di programmazione in quanto il metodo considerato basa il suo funzionamento su un aumento dei gradi di libertà negli elementi attraversati da una o più interfacce.
A questo punto è stato possibile implementare delle classi aventi lo scopo di costruire il sistema lineare associato al problema discreto. Infine ci si è preoccupati di come esportare e visualizzare la soluzione.
Al termine della fase di implementazione sono stati eseguiti dei test per validare il codice, sia da un punto di vista di performance di calcolo sia dal punto di vista della convergenza della soluzione approssimata.
Avendo verificato su un caso test i principali risultati teorici relativi al metodo, abbiamo potuto validare l'implementazione fatta. In particolare si è analizzato l'ordine di convergenza del metodo e si è studiato il condizionamento del problema al variare di alcuni parametri geometrici. I risultati numerici sono in linea con quanto previsto dalle teoria, in particolare si è verificato che il metodo converge quadraticamente in norma L_2 rispetto al passo di griglia.
Dal punto di vista delle performance, l'analisi ha messo in luce alcuni punti in cui è possibile intervenire per rendere il codice più performante.
Le scelte implementative fatte hanno sempre mirato a garantire il funzionamento del codice in parallelo e questo ha rappresentato una difficoltà aggiuntiva. L'analisi di scalabilità ef-

fettuata sul calcolatore LAGRANGE al Cilea ha permesso di validare anche questo aspetto dell'implementazione, anche se in questa direzione molto si può ancora fare per migliorare le prestazioni.

Diversamente da quanto presentato in [14], che tratta problemi bidimensionali, questo lavoro si è interessato all'implementazione del metodo per un problema in un dominio 3D. Questo ha introdotto alcune difficoltà aggiuntive soprattutto per quanto riguarda la parte geometrica del doppio taglio. In realtà, tutta la gestione di una griglia attraversata da più di un'interfaccia (geometria, assegnazione dei gradi di libertà, integrazione etc...) rappresenta un tentativo di ampliamento del lavoro di Hansbo e Hansbo.

Questo lavoro si presta a numerose possibilità di sviluppo. Innanzitutto, nell'ambito degli sviluppi della parte geometrica, estendere l'implementazione in modo che l'interfaccia possa essere definita da una griglia piuttosto che da una funzione *level set*, può essere di grande interesse, in quanto permetterebbe di effettuare calcoli di tipo XFEM su geometrie importate dal mondo reale.
Un'altra possibile strada è quella di considerare altre condizioni di interfaccia, ad esempio permettere soluzioni discontinue, e implementare altri operatori in modo da risolvere problemi differenziali più complessi come, per esempio, il problema di Stokes o l'equazione di Darcy. Questo vuol dire intervenire sulle strutture che creano le matrici con cui viene costruito il sistema globale, arricchendole di nuovi metodi per definire altri operatori differenziali nel caso XFEM.
Visti i risultati presentati circa la convergenza del metodo per il caso del doppio *level set* un interessante ambito di ricerca, anche dal punto di vista teorico, è quello dell'analisi di un problema in cui vi siano due interfacce che intersecano il dominio, risolto con il metodo presentato in [14]. Sul fronte teorico si può innanzitutto valutare se la presenza della doppia interfaccia causa un deterioramento delle proprietà di convergenza del metodo. Interessante inoltre, sia da un punto di vista teorico che numerico, è la stima del parametro di penalità e dei pesi utilizzati nel calcolo dei termini di interfaccia in modo da migliorare la convergenza del metodo.
Infine, l'ottimizzazione del codice è un altro aspetto che merita di essere menzionato come possibile sviluppo. Si sono mostrati i risultati di scalabilità e sono stati evidenziati alcuni comportamenti non pienamente soddisfacenti. Un'analisi più approfondita potrebbe mettere in luce su quali parti del codice bisogna agire. Al di là di questa analisi, un punto critico evidenziato e quindi da migliorare è la creazione della griglia tagliata e l'esportazione della soluzione. Un ulteriore ambito in cui è possibile intervenire è quello del calcolo delle intersezioni, in particolare per quanto riguarda il doppio *level set* dove l'uso di algoritmi ottimizzati potrebbe portare ad un miglioramento del codice.
In conclusione, il codice prodotto è una solida base per ulteriori sviluppi e potrà permettere di studiare un'ampia classe di problemi dove simulare la presenza di superfici di discontinuità è essenziale.

Bibliografia

[1] I. Babuska. "The finite element method for elliptic equations with discontinuous coefficients". In: *Computing* 5/207-213 (1970).

[2] C.B. Barber, D.P. Dobkin e H.T. Huhdanpaa. "The Quickhull algorithm for convex hulls". In: *ACM Transactions on Mathematical Software* 22(4)/469-483 (1996).

[3] John Barret e Charles Elliot. "Fitted and unfitted finite-element methods for elliptic equations with smooth interfaces". In: *Journal of Numerical Analysis* 7/283-300 (1987).

[4] R. Becker e R. Rannacher. "A feed-back approach to error control in finite element methods: basic analysis and examples". In: *East-West J. Numer. Math* 4 (1996), pp. 237–264.

[5] Laura Cattaneo e Claudia Colciago. "Modelli computazionali per la crescita tissutale". Tesi di laurea mag. Politecnico di Milano, 2010.

[6] Z. Chen e J. Zhou. "Finite element methods and thier convergence for elliptic and parabolic interface problems". In: *Numer. Math.* 79/175-202 (1998).

[7] Carlo D'Angelo e Anna Scotti. *A Mixed Finite Element Method for Darcy Flow in Fractured Porous Media with non-matching Grids*. Rapp. tecn. 40. Laboratorio MOX-Politecnico di Milano, 2010.

[8] Christophe Daux et al. "Arbitrary branched and intersecting cracks with the extended finite element method". In: *International Journal for Numerical Methods in Engineering* 48/1741–1760 (2000).

[9] G. Fourestey e S. Deparis. *`LifeV` User Manual*. 2010. URL: `www.lifev.org`.

[10] G. Fourestey et al. *`LifeV` Developer Manual*. 2010. URL: `www.lifev.org`.

[11] Thomas-Peter Fries e Ted Belytscho. "The extended/generalized finite element method: an overview of the method and its applications". In: *International Journal for Numerical Methods in Engineering* 84/253-034 (2010).

[12] A. Gerstenberger e Wolfgang A. Wall. "An eXtended Finite Element Method/Lagrange multiplier based approach for fluid–structure interaction". In: *Comput. Methods Appl. Mechanics Engrg.* 197/19-21 (2007).

[13] A. Hansbo e P. Hansbo. "A finite element method for the simulation of strong and weak discontinuities in solid mechanics". In: *Comput. Methods Appl. Mech. Engrg.* 193/33-35 (2004).

[14] A. Hansbo e P. Hansbo. “An unfitted finite element method, based on Nitsche’s method, for elliptic interface problems”. In: *Comput. Methods Appl. Mech. Engrg.* 191/47-48 (2002).

[15] A. Henderson. *ParaView Guide, A Parallel Visualization Application*. 2007. URL: `http://www.paraview.org/`.

[16] Michael A. Heroux e James M. Willenbring. *Trilinos Users Guide*. Rapp. tecn. SAND2003-2952. Sandia National Laboratories, 2003.

[17] G. Karypis, K. Schloegel e V. Kumar. *`ParMetis`: Parallel Graph Partitioning and Sparse Matrix Ordering*. 2003. URL: `http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview`.

[18] C. Malossi e S. Deparis. *`LifeV` Development Guidelines*. 2011. URL: `www.lifev.org`.

[19] Stanley Osher e Ronald Fedwik. *Level Set Methods and Dynamic Implicit Surfaces*. Springer-Verlag, 2003.

[20] Alfio Quarteroni. *Modellistica numerica per problemi differenziali*. UNITEXT-Springer, 2008.

[21] M. Sala e M. Heroux. *Robust Algebraic Preconditioners with IFPACK 3.0*. Rapp. tecn. SAND-0662. Sandia National Laboratories, 2005.

[22] Sando Salsa. *Equzioni a derivate parziali: metodi, modelli e applicazioni*. UNITEXT-Springer, 2004.

[23] M. Stolarska et al. “Modelling crack growth by level sets in the extended finite element method”. In: *International Journal for Numerical Methods in Engineering* 51/943-960 (2001).

[24] N. Sukumar. “Meshless Methods and Partition of Unity Finite Elements”. In: *In of the Sixth International ESAFORM Conference on Material Forming (2003)*. 2003, pp. 603–606.

[25] *`Git`*. URL: `http://git-scm.com/`.

[26] D. Yang. *C++ and Object Oriented Numeric Computing for Scientists and Engineers*. Springer-Verlag, 2001.

Ringraziamenti

Ringrazio il relatore, Luca Formaggia, per avermi proposto un progetto di tesi stimolante che ha rappresentato una degna conclusione del percorso di studi fatto. Lo ringrazio inoltre per la disponibilità e i consigli dispensati nel corso di questi mesi. Fondamentale è stata anche l'attenzione mostrata nel tener conto delle esigenze del mio lavoro di implementazione nel corso dello sviluppo del *branch RegionMesh*, sul quale è basata gran parte di questa tesi.

Ringrazio il correlatore, Alessio Fumagalli, per avermi seguito nella scoperta di `LifeV` e dei numerosi strumenti utilizzati nel corso di questo lavoro di tesi, senza ovviamente dimenticare la pazienza con cui si è prestato a rispondere ai numerosi dubbi sorti in fase di implementazione.

Ringrazio Alessio (vedi sopra), Anna, Antonio, Franco, Ilaria e Nur, compagni di tender, per avermi supportato e sopportato in questo periodo di tesi. Al di là delle numerose volte in cui mi hanno dato una mano, la loro compagnia ha sicuramente reso questi mesi più interessanti e divertenti.

Ringrazio Chiara, per questi quasi due anni insieme e per avermi incoraggiato e spronato in ogni momento, comprese tutte le volte in cui non sembrava non funzionare niente e vedevo il traguardo della Laurea come un miraggio.

Ringrazio tutta la mia famiglia, che mi è stata vicino in questi anni di Laurea Specialistica (e anche prima, ovviamente).