POLITECNICO DI MILANO

Corso di Laurea in Ingegneria Matematica

TESI DI LAUREA SPECIALISTICA

MODELLAZIONE E SIMULAZIONE NUMERICA DELLE EQUAZIONI DI NAVIER–STOKES CON INTERFACCIA POROSA

Relatore: Chiar.mo Prof. Fabio Nobile
Correlatore: Chiar.mo Prof. Luca Formaggia

Marco Carrettoni: Matr. 735790
Paolo Cravedi : Matr. 739714

ANNO ACCADEMICO 2010-2011

Indice

1 Introduzione **4**

2 Motivazioni e strumenti utilizzati **6**
2.1 LifeV 8
2.2 Strumenti utilizzati 8

3 Modello matematico e discretizzazione numerica **10**
3.1 Il modello matematico 10
3.2 Riformulazione come sottoproblemi accoppiati 11
3.3 Formulazione debole del problema 12
3.4 Buona posizione del problema 13
3.5 Approssimazione ad elementi finiti 14
3.6 Recupero della continuità della velocità 15

4 Aspetti implementativi **20**
4.1 Come funziona LifeV 20
4.2 L'introduzione delle flag 21
4.3 Lettura dei dati in input 24
4.3.1 Regole basilari per l'utilizzo del codice 24
4.3.2 MeshElement.hpp 25
4.3.3 RegionMesh3D.hpp 25
4.3.4 ImporterMesh3D.hpp 31
4.3.5 ImporterMesh3D.cpp 36
4.3.6 MeshPartitioner.hpp 37
4.3.7 BCHandler.hpp 39
4.4 Implementazione di uno spazio discontinuo 40
4.4.1 DOF.hpp 40
4.4.2 DOF.cpp 47
4.4.3 FESpace.hpp 47
4.4.4 MapEpetra.hpp 49
4.5 Il solutore 50
4.5.1 OseenSolver.hpp 50
4.6 Output dei risultati 52

4.6.1 Exporter.hpp 52
4.6.2 ExporterHDF5.hpp 52

5 Risultati numerici **56**
5.1 Cilindro con interfaccia porosa 56
5.2 Un modello di aneurisma 61
5.3 Un'analisi di convergenza 62

6 Conclusioni **66**
6.1 Sviluppi futuri 67

Elenco delle figure

2.1 Geometria di uno stent 6

3.1 Dominio 10
3.2 Sparsità matrice 19

5.1 Mesh: cilindro con interfaccia 56
5.2 Pressione senza interfaccia 57
5.3 Velocità senza interfaccia 57
5.4 Pressione con interfaccia 58
5.5 Velocità con interfaccia 58
5.6 Traiettorie all'interno del cilindro 59
5.7 Pressione al variare di R 59
5.8 P1Bolla-P1 senza fissione all'interfaccia 60
5.9 Andamento della pressione 60
5.10 Modello semplificato di aneurisma cerebrale: mesh 61
5.11 Flusso sanguigno in aneurisma 61
5.12 Caso test: soluzione esatta 63
5.13 Caso test: grafici di convergenza 64

Capitolo 1

Introduzione

Il nostro lavoro si occupa della modellizzazione e della simulazione numerica di un fluido incomprimibile governato dalle equazioni di Navier–Stokes all'interno di un dominio caratterizzato dalla presenza di una parete di interfaccia porosa al suo interno.

La situazione appena descritta può rappresentare il caso di flusso sanguigno all'interno di un aneurisma cerebrale in presenza di uno *stent*. Per ragioni che vedremo spiegate più in dettaglio nel capitolo 2, un possibile approccio semplificato per modellizzare la presenza dello stent consiste nel trascurare i dettagli geometrici delle maglie e considerarlo come un'interfaccia porosa omogenea immersa nel fluido. Dal punto di vista numerico, la presenza dello stent può essere rappresentata per mezzo di un termine dissipativo di superficie aggiunto alle classiche equazioni di Navier–Stokes per flussi incomprimibili.

Tale termine aggiuntivo introduce un salto negli sforzi e, dunque, nella pressione in corrispondenza dell'interfaccia; ciò può essere fonte di problemi dal punto di vista numerico.

Per rispondere a tali problematiche abbiamo proposto l'uso di particolari spazi ad elementi finiti per la pressione; usiamo in particolare elementi continui ovunque, tranne che in corrispondenza della parete di interfaccia. Tale effetto è ottenuto mediante il raddoppio dei gradi di libertà presenti sulla parete interna al dominio su cui poniamo il termine dissipativo.

L'introduzione di un problema modello e della sua discretizzazione a livello numerico sono contenuti nel capitolo 3.

Il nostro lavoro è stato principalmente quello di implementare un codice di calcolo numerico in grado di risolvere problemi di Navier–Stokes in presenza di interfacce porose. Abbiamo deciso di lavorare su LifeV, una libreria di calcolo scientifico basata sul linguaggio C++ sviluppata dal Politecnico di Milano in collaborazione con altri grandi istituti internazionali. La fase di adattamento del codice preesistente di LifeV alle nostre esigenze ha richiesto un grande sforzo implementativo, dato che la versione originale non prevede

in alcun modo l'inserimento di condizioni di tipo Robin (ovvero condizioni che legano la velocità al salto dello sforzo normale) su elementi interni al dominio di calcolo. Tutte le modifiche che abbiamo dovuto apportare ai codici di LifeV sono riportate nel capitolo 4.

Infine abbiamo utilizzato il nostro codice per effettuare alcune simulazioni numeriche e testarne l'efficacia.

Inizialmente abbiamo analizzato il caso di flusso all'interno di un cilindro con presenza di interfaccia porosa. Abbiamo confrontato i risultati ottenuti nel nostro caso test con la soluzione classica di flusso di Poiseuille che ritroveremmo in assenza di condizioni di interfaccia. Abbiamo inoltre confrontato le prestazioni degli spazi ad elementi finiti da noi proposti rispetto agli spazi tradizionali.

Infine abbiamo testato il nostro metodo su un caso test analitico di cui è nota la soluzione esatta e ne abbiamo studiato le proprietà di convergenza.

Tutti i test numerici riportati nella sezione 5 fanno uso di spazi del tipo P1Bolla-P1 con pressione discontinua in corrispondenza dell'interfaccia porosa.

Capitolo 2

Motivazioni e strumenti utilizzati

Per questo lavoro abbiamo preso ispirazione dall'articolo di A. Caiazzo, M.A. Fernandez, J.-F. Gerbeau, V. Martin, *Projection Schemes for Fluid Flows Through a Porous Interface*, pubblicato nel 2008 sulla rivista ESAIM.

Questo articolo si occupa della simulazione numerica di un nuovo congegno medico, chiamato *stent*. Tale dispositivo medico viene utilizzato per il trattamento di aneurismi cerebrali.

Uno stent è costituito da una griglia metallica immersa nel flusso sanguigno. Nelle applicazioni più comuni esso è posizionato a contatto con la parete del vaso e il suo scopo è quello di tenere aperte le arterie in modo da non occludere il flusso sanguigno. Nel caso di aneurismi cerebrali, invece, lo scopo dello stent è differente: esso viene utilizzato per ridurre il flusso all'interno dell'aneurisma e creare un coagulo che possa poi essere rimosso mediante un'operazione chirurgica. Lo stent viene dunque inserito in questo caso all'interno del flusso sanguigno.

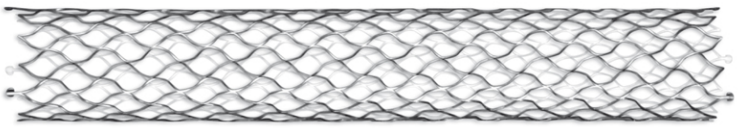

Figura 2.1: Esempio di geometria di uno stent.

La simulazione numerica è necessaria per quantificare la riduzione di vorticità e stress alle pareti dell'aneurisma dovuta alla presenza dello stent e i suoi effetti sulla circolazione sanguigna nelle zone vicine.

Il tipo di stent utilizzato per aneurismi cerebrali è una sorta di rete metallica costituita da fili molto sottili (circa 40 μm), con finestre molto piccole (100 μm) e una struttura multistrato. La parte a contatto con l'aneurisma è caratterizzata da una griglia molto fine in modo da garantire la riduzione del flusso sanguigno e la formazione di un trombo nell'aneurisma.

In generale tutti i lavori di simulazione numerica di uno stent finora effettuati prevedono la rappresentazione accurata delle maglie dello stent all'interno della mesh, in modo da cogliere opportunamente il comportamento del fluido durante l'attraversamento della griglia. Date le piccolissime dimensioni delle maglie dello stent, una buona simulazione numerica richiede un grande sforzo per generare una mesh adatta alle esigenze e, come appare ovvio, un elevatissimo onere computazionale.

L'articolo da cui abbiamo preso ispirazione per il nostro lavoro propone una possibile soluzione per queste problematiche. Dato lo spessore infinitesimo dello stent, possiamo modellizzare quest'ultimo come una superficie di mezzo poroso immersa nel fluido. Rappresentiamo dunque lo stent solo a livello macroscopico, trascurandone in tal modo ogni caratteristica fisica: deformazione della struttura, interazione con la parete del vaso, ecc. Non è più necessario dunque utilizzare una mesh così fine da cogliere ogni caratteristica di ogni singola maglia; basta una griglia molto più lasca, che sia conforme alla superficie di interfaccia interna al dominio che corrisponde alle pareti dello stent.

Dal punto di vista matematico, il comportamento del fluido è modellizzato per mezzo delle equazioni di Navier–Stokes incomprimibili, mentre la presenza dello stent è riprodotta mediante l'aggiunta sull'interfaccia di una condizione di Robin di tipo resistivo.

Tale termine addizionale di tipo dissipativo introduce un salto negli sforzi attraverso la superficie dello stent, che può essere fonte di problemi numerici: un'approssimazione continua della pressione può portare a risultati non accurati, mentre un'approssimazione totalmente discontinua porta a simulazioni molto costose.

L'idea proposta dall'articolo è quella di utilizzare uno spazio ad elementi finiti per la pressione che sia continuo ovunque, tranne in corrispondenza dell'interfaccia. In particolare, i gradi di libertà in corrispondenza dell'interfaccia saranno raddoppiati.

Prendendo spunto da quanto presentato nell'articolo, abbiamo deciso di implementare un codice di calcolo numerico che fosse in grado di risolvere problemi di Navier–Stokes incomprimibili all'interno di un dominio con condizione di Robin imposta su un'interfaccia interna.

Per gli aspetti implementativi del nostro lavoro abbiamo utilizzato il codice di calcolo *LifeV*, una libreria per il calcolo numerico sviluppata da una collaborazione tra il Politecnico di Milano e altre tre importanti università mondiali. Lo sforzo richiesto dalla fase di programmazione è stato molto pesante; sono state necessarie molte modifiche al codice originale di LifeV,

dato che questo non prevedeva in origine la possibilità di imporre condizioni su interfacce interne al dominio di calcolo. È stato necessario agire su più livelli: lettura della mesh, imposizione delle condizioni al contorno, solutore, esportazione dei risultati.

Abbiamo poi utilizzato il nostro codice per risolvere un caso test di flusso all'interno di un cilindro con parete interna di interfaccia su cui abbiamo imposto una condizione di Robin di tipo resistivo.

Per la risoluzione abbiamo utilizzato spazi P1Bolla per la velocità e P1 per la pressione.

In seguito, abbiamo svolto un'analisi di convergenza del nostro codice su un caso test in cui è nota la soluzione esatta.

2.1 LifeV

Il nostro lavoro si è basato su codici già esistenti. In particolare abbiamo utilizzato come base i codici contenuti all'interno della libreria LifeV.

LifeV è una libreria di elementi finiti che fornisce l'implementazione in C++ dei più avanzati metodi matematici e numerici attualmente in uso. Essa è utilizzata sia per scopi di ricerca sia per necessità di simulazioni numeriche in vari campi. Ad esempio LifeV è stato già usato in diversi contesti medici ed industriali, in particolare nell'ambito di simulazioni di interazioni fluido–struttura e di trasporto di massa.

LifeV è il frutto della collaborazione di quattro istituzioni: École Polytechnique Fédérale de Lausanne (CMCS) in Svizzera, Politecnico di Milano (MOX) in Italia, INRIA (REO, ESTIME) in Francia e Emory University (Sc. Comp.) negli Stati Uniti.

Lo sviluppo della libreria ha avuto inizio nel 2002 dalla collaborazione delle prime tre università; ad oggi la libreria di LifeV è arrivata a comprendere oltre 50000 linee di codice.

LifeV è attualmente open–source ed è scaricabile gratuitamente dal sito `http://www.lifev.org` previa richiesta di licenza.

Nel 2010 è stata resa disponibile una nuova release del codice nella quale è stata fornita una versione parallela dell'intera libreria che garantisse alte prestazioni di calcolo su architetture a multiprocessori. Il numero più alto di gradi di libertà gestiti attualmente in una simulazione con LifeV è pari a nove milioni.

Al momento tutti gli sforzi sono orientati a rendere LifeV capace di sfruttare al massimo le capacità dei nuovi supercalcolatori.

2.2 Strumenti utilizzati

Una grande parte del tempo impiegato per questo progetto è stata dedicata ad apprendere come utilizzare gli strumenti necessari. In particolare è stato

grande lo sforzo richiesto per riuscire a comprendere a fondo le basi per l'utilizzo di LifeV.

Per la visualizzazione grafica dei risultati delle simulazioni abbiamo utilizzato il programma ParaView. ParaView è un programma open–source disponibile gratuitamente per la visualizzazione grafica di simulazioni numeriche, anche in parallelo. Permette visualizzazioni anche in 3D, è adatto a gestire grandi insiemi di dati ed è multi–piattaforma, cioè in grado di girare su più sistemi operativi. Una sua caratteristica è l'interfaccia, che è molto intuitiva e semplice da utilizzare per l'utente.

Per la generazione di mesh abbiamo inoltre utilizzato il programma TetGen. TetGen è un generatore di mesh tridimensionali: è in grado di generare mesh di tetraedri adatte a simulazioni numeriche.

Dato il bordo di un dominio tridimensionale, Tetgen, tramite un processo detto triangolazione di Delauney, è in grado di costruire dapprima la griglia di bordo, poi quella completa in base ad alcuni parametri fissati.

TetGen permette in pratica di generare una griglia tetraedrica di qualità conforme ad un certo dominio sotto alcuni vincoli di dimensione e forma degli elementi che la compongono.

Per garantire la buona qualità della mesh, TetGen utilizza una generalizzazione in tre dimensioni del raffinamento di Delauney. Esso consiste in un raffinamento di una griglia tramite l'aggiunta di vertici; il posizionamento di tali vertici aggiuntivi permette di ottenere mesh più dettagliate e di miglior qualità.

Capitolo 3

Modello matematico e discretizzazione numerica

3.1 Il modello matematico

Introduciamo ora un modello matematico che rappresenti la situazione che vogliamo considerare: un flusso rappresentato dalle equazioni di Navier–Stokes incomprimibili con condizione di tipo resistivo su un'interfaccia interna al dominio di calcolo.

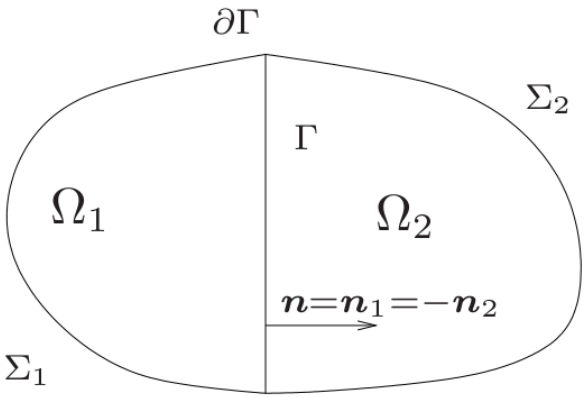

Figura 3.1: Dominio di calcolo per il modello in esame.

Sia Ω un dominio semplicemente connesso di $\mathbb{R}^n, n = 2, 3$ e $\partial\Omega$ il suo bordo regolare. Una rappresentazione del dominio Ω è fornita in figura 3.1. Supponiamo che lo stent sia rappresentato da una superficie regolare Γ immersa in Ω.

Siano inoltre $\mathbf{u}$ e p la velocità e la pressione del fluido e definiamo i tensori $\mathbf{D}$ e $\mathbf{T}$:

$$\mathbf{D}(\mathbf{u}) = \frac{1}{2}(\nabla\mathbf{u} + \nabla\mathbf{u}^T), \qquad \mathbf{T}(\mathbf{u}, p) = -p\mathbf{I} + 2\mu\mathbf{D}(\mathbf{u})$$

dove $\mathbf{T}(\mathbf{u}, p)$ è detto tensore degli sforzi di Cauchy, μ è la viscosità dinamica del fluido e $\mathbf{I}$ è la matrice identità di $\mathbb{R}^n$.

Supponiamo che il fluido sia descritto dalle equazioni di Navier–Stokes incomprimibili e modellizziamo lo stent con un termine dissipativo di superficie che aggiungiamo all'equazione di conservazione del momento. Il modello è dunque il seguente:

$$\begin{cases} \rho(\frac{\partial\mathbf{u}}{\partial t} + \mathbf{u}\cdot\nabla\mathbf{u}) - \mathrm{div}\mathbf{T}(\mathbf{u}, p) + \mathbf{R}_\Gamma\mathbf{u}\delta_\Gamma = \mathbf{f} & \text{in } \Omega \\ \mathrm{div}\mathbf{u} = 0 & \text{in } \Omega \end{cases} \tag{3.1}$$

dove ρ è la densità del fluido, $\mathbf{f}$ è un termine sorgente, $\mathbf{R}_\Gamma$ è un tensore simmetrico e definito positivo che rappresenta la dissipazione dovuta alla presenza dello stent e δ_Γ è la misura di Dirac sulla superficie Γ. Presa $\mathbf{v} \in [H^1(\Omega)]^n$ si ha in pratica

$$\langle \mathbf{R}_\Gamma\mathbf{u}\delta_\Gamma, \mathbf{v}\rangle = \int_\Gamma \mathbf{R}_\Gamma\mathbf{u}\cdot\mathbf{v}\, d\Gamma$$

3.2 Riformulazione come sottoproblemi accoppiati

Assumiamo ora che l'interfaccia porosa Γ suddivida il dominio del fluido in due sottodomini connessi tali che

$$\Omega = \Omega_1 \cup \Gamma \cup \Omega_2, \qquad \Sigma_i := \partial\Omega \cap \bar{\Omega}_i,\ i = 1, 2$$

In ciascun sottodominio definiamo la normale uscente dall'interfaccia, rispettivamente $\mathbf{n}_1$ e $\mathbf{n}_2$, e definiamo $\mathbf{n} := \mathbf{n}_1 = -\mathbf{n}_2$.

Il problema che dobbiamo considerare è il seguente

$$\begin{cases} \rho(\frac{\partial\mathbf{u}}{\partial t} + \mathbf{u}\cdot\nabla\mathbf{u}) - \nabla p - 2\mu\mathrm{div}\mathbf{D}(\mathbf{u}) + \mathbf{R}_\Gamma\mathbf{u}\delta_\Gamma = \mathbf{f} & \text{in } \Omega \\ \mathrm{div}\mathbf{u} = 0 & \text{in } \Omega \\ \mathbf{u} = \mathbf{0} & \text{su } \partial\Omega \end{cases} \tag{3.2}$$

dove $\mathbf{R}_\Gamma$ è la resistenza all'interfaccia, legata alla permeabilità e alla porosità del materiale. Infine, assumiamo condizioni al bordo di Dirichlet omogenee su $\partial\Omega$. Se per qualsiasi campo q definito su Ω indichiamo con la notazione $q_i := q|_{\Omega_i}$ la sua restrizione sul sottodominio Ω_i, possiamo definire i seguenti salti attraverso Γ:

$$\begin{gathered} [[\mathbf{u}]] = \mathbf{u}_{1|\Gamma} - \mathbf{u}_{2|\Gamma} \\ [[\mathbf{D}(\mathbf{u})\mathbf{n}]] = \mathbf{D}(\mathbf{u}_1)_{|\Gamma}\mathbf{n}_1 + \mathbf{D}(\mathbf{u}_2)_{|\Gamma}\mathbf{n}_2 \\ [[p\mathbf{n}]] = p_{1|\Gamma}\mathbf{n}_1 + p_{2|\Gamma}\mathbf{n}_2 \end{gathered} \tag{3.3}$$

Possiamo dunque riformulare il nostro problema come due sottoproblemi accoppiati

$$\begin{array}{ll} \rho(\frac{\partial \mathbf{u}_i}{\partial t} + \mathbf{u}_i \cdot \nabla \mathbf{u}_i) - \nabla p_i - 2\mu \text{div}\mathbf{D}(\mathbf{u}_i) = \mathbf{f}_i & \text{in } \Omega_i \\ \text{div}\mathbf{u}_i = 0 & \text{in } \Omega_i \\ \mathbf{u}_i = \mathbf{0} & \text{su } \Sigma_i \end{array} \tag{3.4}$$

con le seguenti condizioni di interfaccia

$$[[\mathbf{u}]] = \mathbf{0} \qquad \text{su } \Gamma \tag{3.5}$$

$$[[\mathbf{T}(\mathbf{u}, p)]] = [[2\mu \mathbf{D}(\mathbf{u})\mathbf{n} - p\mathbf{n}]] = -R_\Gamma \mathbf{u} \qquad \text{su } \Gamma \tag{3.6}$$

Queste condizioni di interfaccia servono ad imporre la continuità della velocità e a collegare il salto del tensore degli sforzi alla velocità stessa.

3.3 Formulazione debole del problema

Forniamo ora la formulazione debole del problema in esame. Cominciamo con la definizione degli opportuni spazi funzionali. Siano:

$$V = \left[H_0^1(\Omega)\right]^3$$

$$Q = L_0^2(\Omega)$$

$$M = \left\{w \ : \ w_i \in L^2(\Omega_i)\right\}$$

$$N = M \cap Q$$

Nel nostro caso si ha $u \in V$ e $p \in N$.

Cerchiamo di ricavare la formulazione debole del problema di partenza. Sia $\mathbf{v} \in V$:

$$\sum_{i=1,2} \left\{ \int_{\Omega_i} \rho \frac{\partial \mathbf{u}_i}{\partial t} \mathbf{v}_i + \int_{\Omega_i} \rho \mathbf{u}_i \cdot \nabla \mathbf{u}_i \mathbf{v}_i + \int_{\Omega_i} \nabla p_i \mathbf{v}_i - \int_{\Omega_i} 2\mu \text{div}(\mathbf{D}(\mathbf{u}_i))\mathbf{v}_i \right\} = \int_\Omega \mathbf{f}\mathbf{v}$$

da cui

$$\begin{array}{l} \sum_{i=1,2} \left\{ \int_{\Omega_i} \rho \frac{\partial \mathbf{u}_i}{\partial t} \mathbf{v}_i + \int_{\Omega_i} \rho \mathbf{u}_i \cdot \nabla \mathbf{u}_i \mathbf{v}_i - \int_{\Omega_i} p_i \text{div}\mathbf{v} + \int_\Gamma p_i \mathbf{n}_i \mathbf{v}_i + \int_{\Omega_i} 2\mu \mathbf{D}(\mathbf{u}_i) \nabla \mathbf{v}_i \right. \\ \left. - \int_\Gamma 2\mu \mathbf{D}(\mathbf{u}_i)\mathbf{n}_i \mathbf{v}_i + \int_{\Sigma_i} p_i \mathbf{n}_i \mathbf{v}_i - \int_{\Sigma_i} 2\mu \mathbf{D}(\mathbf{u}_i) n_i \mathbf{v}_i \right\} = \int_\Omega \mathbf{f}\mathbf{v} \end{array}$$

Tenendo conto delle condizioni al bordo e di interfaccia e svolgendo la sommatoria si ottiene:

$$\begin{array}{l} \int_\Omega \rho \frac{\partial \mathbf{u}}{\partial t} \mathbf{v} + \int_\Omega \rho \mathbf{u} \cdot \nabla \mathbf{u} \mathbf{v} - \int_\Omega p \text{div}\mathbf{v} + \int_\Omega 2\mu \mathbf{D}(\mathbf{u}) \cdot \nabla \mathbf{v} + \int_\Gamma (p_1 \mathbf{n}_1 + p_2 \mathbf{n}_2 \\ -2\mu \mathbf{D}(u_1)\mathbf{n}_1 - 2\mu \mathbf{D}(\mathbf{u}_2)\mathbf{n}_2)\mathbf{v} = \int_\Omega \mathbf{f}\mathbf{v} \end{array} \tag{3.7}$$

Usando la seconda condizione di interfaccia:

$$p_1\mathbf{n}_1 + p_2\mathbf{n}_2 - 2\mu\mathbf{D}(\mathbf{u}_1)\mathbf{n}_1 - 2\mu\mathbf{D}(\mathbf{u}_2)\mathbf{n}_2 = \mathbf{R}_\Gamma\mathbf{u}$$

si ha

$$\int_\Omega \rho\frac{\partial\mathbf{u}}{\partial t}\mathbf{v} + \int_\Omega \rho\mathbf{u}\cdot\nabla\mathbf{u}\mathbf{v} - \int_\Omega p\mathrm{div}\mathbf{v} + \int_\Omega 2\mu\mathbf{D}(\mathbf{u})\cdot\nabla\mathbf{v} + \int_\Gamma \mathbf{R}_\Gamma\mathbf{u}\mathbf{v} = \int_\Omega \mathbf{f}\mathbf{v}$$

ovvero

$$(\rho\frac{\partial\mathbf{u}}{\partial t},\mathbf{v}) + (\rho\mathbf{u}\cdot\nabla\mathbf{u},\mathbf{v}) - (p,\mathrm{div}\mathbf{v}) + (2\mu\mathbf{D}(\mathbf{u}),\nabla\mathbf{v}) + (\mathbf{R}_\Gamma\mathbf{u},\mathbf{v})_\Gamma = (\mathbf{f},\mathbf{v}) \quad (3.8)$$

Per quanto riguarda la seconda equazione del problema (3.1), si ottiene semplicemente, prendendo $q \in N$:

$$\int_\Omega \mathrm{div}\mathbf{u}\, q = 0 \quad (3.9)$$

La formulazione debole del problema diventa dunque: trovare $\mathbf{u} \in V$, $p \in N$ tali che, per ogni funzione test $v \in V$, $q \in N$

$$\begin{cases} a(\mathbf{u},\mathbf{v}) + c(\mathbf{u},\mathbf{u},\mathbf{v}) + b(\mathbf{v},p) = F(\mathbf{v}) \\ b(\mathbf{u},q) = 0 \end{cases} \quad (3.10)$$

con

$$\begin{array}{l} a(\mathbf{u},\mathbf{v}) = 2\mu(\mathbf{D}(\mathbf{u}),\mathbf{D}(\mathbf{v})) + (\mathbf{R}_\Gamma\mathbf{u},\mathbf{v})_\Gamma \\ b(\mathbf{v},p) = -(\mathrm{div}\mathbf{v},p) \\ c(\mathbf{w},\mathbf{u},\mathbf{v}) = (\mathbf{w}\cdot\nabla\mathbf{u},\mathbf{v}) \end{array} \quad (3.11)$$

3.4 Buona posizione del problema

Per semplificare l'analisi del problema facciamo due importanti assunzioni:

1. il fluido è governato dalle equazioni di Stokes stazionarie;
2. lo stent è rappresentato da una superficie Γ che suddivide il dominio Ω in due sottodomini connessi

$$\Omega_f = \Omega \setminus \Gamma = \Omega_1 \cup \Omega_2, \qquad \Omega_1 \cap \Omega_2 = \emptyset$$

Utilizziamo le notazioni introdotte in precedenza e definiamo

$$M = L^2(\Omega), \qquad Q = L^2_0(\Omega), \qquad V = [H^1_0(\Omega)]^3$$

con le loro usuali norme $\|\cdot\|_{0,\Omega}$ e $\|\cdot\|_{1,\Omega}$.

Il corrispettivo del problema (3.1) per il problema di Stokes è dato dalla seguente formulazione a due domini:

$$\begin{array}{ll} -\mu \mathrm{div}\mathbf{u}_i + \nabla p_i = \mathbf{f}_i & \text{in } \Omega_i,\ i = 1,2 \\ \mathrm{div}\mathbf{u}_i = 0 & \text{in } \Omega_i,\ i = 1,2 \\ \mathbf{u_1} = \mathbf{u_2} & \text{su } \Gamma \\ \mu\nabla\mathbf{u}_1 \cdot \mathbf{n}_1 - p_1\mathbf{n}_1 + \mu\nabla\mathbf{u}_2 \cdot \mathbf{n}_2 - p_2\mathbf{n}_2 = -\mathbf{R}_\Gamma\mathbf{u} & \text{su } \Gamma \\ \mathbf{u}_i = 0 & \text{in } \Sigma_i,\ i = 1,2 \end{array} \tag{3.12}$$

Per comodità abbiamo scelto condizioni di bordo di Dirichlet omogenee sul bordo esterno del dominio.

La formulazione debole, analogamente a quanto visto in precedenza, diventa:

$$\sum_{i=1}^{2} [(\mu\nabla\mathbf{u}_i, \nabla\mathbf{v}_i)_{\Omega_i} - (p_i, \mathrm{div}\mathbf{v}_i)_{\Omega_i} - (\mu\nabla\mathbf{u}_i \cdot \mathbf{n}_i - p_i\mathbf{n}_i, \mathbf{v}_i)_\Gamma] = (\mathbf{f}, \mathbf{v})_\Omega$$

Per la regolarità di $\mathbf{u}$ e $\mathbf{v}$ e per la quarta equazione del sistema precedente otteniamo

$$\begin{cases} (\mu\nabla\mathbf{u}, \nabla\mathbf{v})_\Omega + (\mathbf{R}_\Gamma\mathbf{u}, \mathbf{v})_\Gamma - (\mathrm{div}\mathbf{v}, p)_\Omega = (\mathbf{f}, \mathbf{v})_\Omega & \forall \mathbf{v} \in V \\ (\mathrm{div}\mathbf{u}, r)_\Omega = 0 & \forall r \in Q \end{cases} \tag{3.13}$$

Possiamo dimostrare facilmente che il problema (3.13) ammette una soluzione unica.

Definiamo infatti la forma bilineare $a(\cdot,\cdot)$ in $V \times V$:

$$a(\mathbf{u}, \mathbf{v}) = (\mu\nabla\mathbf{u}, \nabla\mathbf{v})_\Omega + (\mathbf{R}_\Gamma\mathbf{u}, \mathbf{v})_\Gamma$$

Tale forma bilineare è continua (conseguenza del teorema di traccia) e coerciva (conseguenza della disuguaglianza di Poincaré e della positività di $\mathbf{R}_\Gamma$). Per la teoria classica delle equazioni di Stokes si dimostra quindi l'esistenza e unicità della velocità. L'esistenza e l'unicità della pressione discendono dalla classica condizione inf-sup associata alla forma bilineare

$$b(\mathbf{v}, \mathbf{q})$$

Per ulteriori dettagli, fare riferimento a [8].

Questi semplici risultati si possono estendere anche al caso più generale del problema (3.1) con equazioni di Navier–Stokes incomprimibili (vedere [2]).

3.5 Approssimazione ad elementi finiti

Per la discretizzazione numerica del problema (3.13) utilizziamo la classica teoria di Galerkin. Sia T_h una triangolazione regolare del dominio Ω, che

supponiamo Lipschitziano. Indichiamo con h il livello di raffinamento della triangolazione

$$h = \max_{T \in T_h} h_T$$

dove h_T è il diametro dell'elemento T della triangolazione. Assumiamo per semplicità che la triangolazione sia quasi–uniforme, ovvero che esistano due costanti C_1 e C_2 tali che

$$C_1 h_T \leq h \leq C_2 h_T \qquad \forall T \in T_h$$

Supponiamo inoltre che la triangolazione T_h sia conforme con l'interfaccia Γ. Ciò significa che esiste un sottoinsieme $G_h \subset T_h$ formato dagli elementi di T_h (facce in 3D, lati in 2D) che costituiscono la superficie di interfaccia.

Data una funzione ϕ definita a tratti e continua, il salto $[[\phi \mathbf{n}]]$ su un lato $E \in G_h$ è definito come

$$[[\phi \mathbf{n}]](\mathbf{x}) = \lim_{t \to 0^+} (\phi(\mathbf{x} + t\mathbf{n}_{1,E})\mathbf{n}_{1,E} + \phi(\mathbf{x} + t\mathbf{n}_{2,E})\mathbf{n}_{2,E})$$

con $\mathbf{n}_{i,E}$ versore normale ad E e uscente da Ω_i

Definiamo a questo punto gli spazi degli elementi finiti che utilizziamo per la risoluzione numerica del nostro problema. Siano:

$$\begin{aligned} V_h^k &= \left\{ \mathbf{v}_h \in C^0(\overline{\Omega}) : \mathbf{v}_h|_T \in (\mathbb{P}_k)^3, \forall T \in T_h \right\} \cap V \\ M_{h,i}^k &= \left\{ q_h \in C^0(\overline{\Omega}_i) : q_h|_T \in \mathbb{P}_k, \forall T \in T_h \right\} \\ M_h^k &= \left\{ q_h \in L^2(\Omega) : q_h|_{\Omega_i} \in M_{h,i}^k, \forall i = 1, 2 \right\} \end{aligned} \tag{3.14}$$

Per le nostre simulazioni useremo uno spazio P1Bolla per la velocità ed uno spazio P1 per le pressioni. Abbiamo scelto questa coppia perchè sappiamo essere stabile nel caso classico senza fissione all'interfaccia. Inoltre, rispetto alle altre coppie stabili, è quella, data l'implementazione di LifeV, che permette tempi di calcolo ragionevoli.

Lo spazio della pressione è continuo ovunque e discontinuo lungo l'interfaccia. L'introduzione della discontinuità avviene per mezzo di un raddoppio dei gradi di libertà situati sull'interfaccia Γ. Per ciascuna coppia grado di libertà originale–grado di libertà ripetuto il primo va assegnato al sottodominio Ω_1, mentre il secondo a Ω_2.

Per praticità di implementazione abbiamo deciso di raddoppiare anche i gradi di libertà relativi alla velocità: ne recupereremo poi la continuità agendo sulle matrici del problema in modo da raccordare i valori della velocità su ogni grado di libertà ripetuto tramite la condizione di salto nullo.

3.6 Recupero della continuità della velocità

Il recupero della continuità della velocità viene fatto raccordando ogni coppia di gradi di libertà ripetuti situata sull'interfaccia.

Per fare ciò abbiamo utilizzato la stessa tecnica che viene adoperata per l'introduzione di condizioni essenziali nel caso di problemi vettoriali. Siamo infatti nel caso in cui la condizione al bordo che vogliamo imporre nella matrice non riguarda una singola componente del vettore delle incognite, bensì una loro combinazione lineare.

Consideriamo ad esempio di voler imporre in un problema di Navier–Stokes una condizione del tipo $\mathbf{u}^T\mathbf{n} = g$ sul bordo Γ_D, dove $\mathbf{n}$ è il versore normale al bordo. Tale condizione coinvolge una combinazione lineare delle componenti di $\mathbf{u}$ e può essere espressa come $u_x n_x + u_y n_y + u_z n_z = g$. Se la normale al bordo è allineata con uno degli assi, ci ritroviamo al caso in cui la condizione è prescritta su una singola componente; nel caso generale in cui non c'è allineamento tra normale e assi cartesiani occorre invece agire su più componenti contemporaneamente.

Il nostro caso non è molto diverso da quello appena descritto. Se consideriamo infatti la coppia nodo originale–nodo ripetuto $(\bar{i},\bar{j})$, dobbiamo imporre la condizione di salto nullo per la velocità. Alla coppia sono dunque associati i vincoli

$$u_{\bar{i},k} = u_{\bar{j},k}, \quad k = x, y, z$$

che possono essere riscritti come combinazione lineare del vettore delle incognite:

$$u_{\bar{i},k} - u_{\bar{j},k} = 0, \quad k = x, y, z$$

Vediamo come si può procedere operativamente per l'imposizione di tali condizioni al bordo.

Supponiamo che $\mathbf{U} \in \mathbb{R}^n$ sia il vettore che contiene tutte le incognite del nostro problema discretizzato, con n pari al numero di gradi di libertà per ciascuna componente della velocità più quello relativo alla pressione. Consideriamo il caso in cui le condizioni che vogliamo imporre possano essere scritte nella forma

$$N^T\mathbf{U} = \mathbf{g} \tag{3.15}$$

con $N \in \mathbb{R}^{n\times m}$. Tornando al nostro problema, imporre il vincolo $u_{\bar{i},k} = u_{\bar{j},k}$ all'interno dalla matrice N significa inserire in tale matrice una colonna contenente tutti zeri ad eccezione di un 1 sulla $\bar{i}$–esima riga e di un -1 sulla $\bar{j}$–esima riga. Per comodità normalizziamo ciascuna colonna in modo che abbia norma unitaria. Questa operazione viene effettuata per ciascuna coppia di gradi di libertà ripetuti e per ciascuna componente della velocità; chiamando dunque r il numero di gradi di libertà ripetuti, si ha che $m = 3{*}r$.

Abbiamo perciò:

$$N = [\overline{n}_1, \overline{n}_2, ...\overline{n}_m], \qquad \overline{n}_i = \frac{1}{\sqrt{2}}[0, 0, 1, ... - 1, ...0, 0]^T$$

da cui

$$N^T N = I$$

Per comodità di notazione usiamo la matrice

$$K = NN^T \in \mathbb{R}^{n\times n}$$

con componenti $K_{ij} = N_i N_j$. La matrice K gode delle seguenti proprietà:

1. è simmetrica.

2. ha rango pari a m. Infatti $K\mathbf{v}$ è la proiezione ortogonale di $\mathbf{v}$ nel sottospazio definito dalle colonne di N, essendo per definizione $K\mathbf{v} = (N^T\mathbf{v})N$. Di conseguenza $K\mathbf{v} = \mathbf{0}$ per ogni vettore $\mathbf{v}$ ortogonale a N, cioè tale che $N^T\mathbf{v} = \mathbf{0}$.

Siano ora A e $\mathbf{b}$ la matrice di massa e il termine noto del nostro problema prima dell'imposizione delle condizioni (3.15). Per imporre queste è possibile utilizzare la tecnica dei moltiplicatori di Lagrange. Essa corrisponde ad aggiungere un ulteriore vettore di incognite $\lambda \in \mathbb{R}^m$ e risolvere il seguente problema vincolato

$$\begin{cases} A\mathbf{U} + N\lambda = \mathbf{b} \\ N^T\mathbf{U} = \mathbf{g} \end{cases} \tag{3.16}$$

Ricordiamo inoltre che nel nostro caso si ha $\mathbf{g} = \mathbf{0}$, per cui la seconda equazione del sistema (3.16) si riduce a

$$N^T\mathbf{U} = \mathbf{0} \tag{3.17}$$

Premoltiplicando la prima equazione per N^T abbiamo

$$N^T A\mathbf{U} + I\lambda = N^T\mathbf{b}$$

quindi

$$\lambda = N^T\mathbf{b} - N^T A\mathbf{U}$$

che sostituita nella prima equazione ci dà

$$(I - K)A\mathbf{U} = \mathbf{b} - K\mathbf{b}$$

Indicata con $Z = (I - K)$ e considerando che $A = A(I - K) + AK$ possiamo scrivere

$$ZAZ\mathbf{U} + ZAK\mathbf{U} = \mathbf{b} - K\mathbf{b} = Z\mathbf{b} \tag{3.18}$$

Facciamo ora la seguente considerazione: il vettore delle incognite $\mathbf{U}$ e il termine noto $\mathbf{b}$ possono essere decomposti nel modo seguente:

$$\mathbf{U} = \mathbf{U}^N + \mathbf{U}^\perp = N\mathcal{U}^N + \mathbf{U}^\perp, \qquad \mathcal{U}^N \in \mathbb{R}^m$$

$$\mathbf{b} = \mathbf{b}^N + \mathbf{b}^\perp = Nb^N + \mathbf{U}^\perp, \qquad b^N \in \mathbb{R}^m$$

dove $\mathbf{U}^N = N\mathcal{U}^N$ e $\mathbf{b}^N = Nb^N$ sono le proiezioni ortogonali nel sottospazio definito dalle colonne di N di $\mathbf{U}$ e $\mathbf{b}$ rispettivamente, mentre $\mathbf{U}^\perp$ e $\mathbf{b}^\perp$

sono le componenti ortogonali a tale sottospazio. Valgono inoltre le identità $\mathbf{U}^N = K\mathbf{U}$, $\mathbf{b}^N = K\mathbf{b}$, $\mathbf{U}^\perp = Z\mathbf{U}$, $\mathbf{b}^\perp = Z\mathbf{b}$.

Dalla formula (3.15) ricaviamo

$$N^T\mathbf{U} = N^T(N\mathcal{U}^N + \mathbf{U}^\perp) = \mathcal{U}^N = \mathbf{g}$$

mentre da 3.18 ricaviamo

$$ZA\mathbf{U}^\perp = \mathbf{b}^\perp - ZAK\mathbf{U} = \mathbf{b}^\perp - ZA\mathbf{U}^N = \mathbf{b}^\perp - ZAN\mathbf{g} \tag{3.19}$$

Per cui il sistema (3.16) è equivalente a

$$\begin{cases} \mathcal{U}^N = \mathbf{g} \\ ZA\mathbf{U}^\perp = Z\mathbf{b} - ZAN\mathbf{g} \end{cases} \tag{3.20}$$

o equivalentemente a

$$\begin{cases} K\mathbf{U} = N\mathbf{g} \\ ZAZ\mathbf{U} = Z\mathbf{b} - ZAN\mathbf{g} \end{cases} \tag{3.21}$$

Possiamo ora prendere una arbitraria combinazione lineare delle due equazioni e riscrivere (3.21) nel problema equivalente

$$ZAZ\mathbf{U} + \alpha K\mathbf{U} = Z\mathbf{b} - (ZA - \alpha I)N\mathbf{g} \tag{3.22}$$

Essendo nel nostro caso $\mathbf{g} = 0$ l'equazione (3.22) si riduce a risolvere

$$ZAZ\mathbf{U} + \alpha K\mathbf{U} = Z\mathbf{b} \tag{3.23}$$

con $\alpha > 0$ arbitrario.

In generale il valore del parametro α non interferisce nella soluzione $\mathbf{U}$ del problema. Esso può essere scelto opportunamente per non mal condizionare il sistema, anche se spesso nella pratica viene scelto $\alpha = 1$.

Nell'operazione di inserimento delle condizioni di continuità della velocità lungo l'interfaccia, il termine ZAZ ha l'effetto di porre nella matrice del sistema su ogni riga e colonna corrispondente a due gradi di libertà accoppiati una media dei due contributi; il termine αK corrisponde invece all'inserimento di un termine diagonale e permette di ovviare alla singolarità della matrice ZAZ.

Siamo dunque in grado di trasformare il sistema originale $A\mathbf{U} = \mathbf{b}$ in un sistema modificato

$$\tilde{A}\mathbf{U} = \tilde{\mathbf{b}}$$

Il pattern di sparsità della matrice $\tilde{A}$ è riportato in figura 3.2.

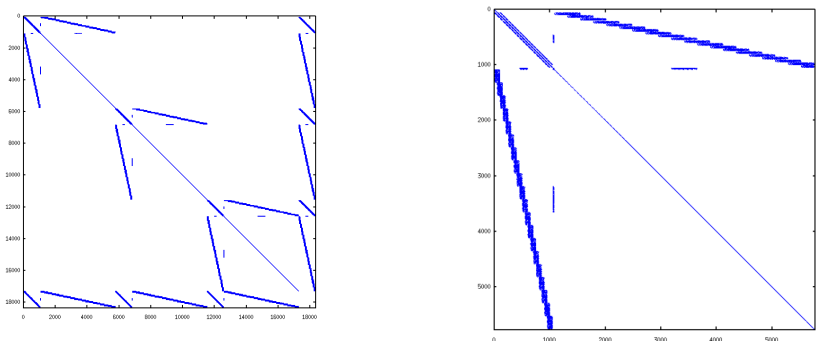

Figura 3.2: Sparsità della matrice per gli elementi P1Bolla-P1 con discontinuità; Dettaglio dei DOF relativi alla componente x della velocità

Capitolo 4

Aspetti implementativi

Il nostro lavoro su LifeV è consistito nell'apportare le opportune modifiche in modo che il codice fosse in grado di accettare condizioni su interfacce interne. Le modifiche hanno interessato una vasta parte del codice, a partire dalla lettura della mesh, passando per la fase di costruzione degli spazi degli elementi finiti, alla soluzione del problema fino all'esportazione dei risultati ottenuti mediante ParaView.

Inoltre, abbiamo riscritto buona parte del codice in modo tale che si potesse utilizzare una recente novità in LifeV, le *flag*; esse hanno lo scopo di definire l'elemento geometrico che stiamo considerando come elemento di bordo, di interfaccia, interno o altro.

Per capire le modifiche effettuate è necessario dapprima capire come in generale LifeV risolve un problema differenziale mediante elementi finiti.

4.1 Come funziona LifeV

Consideriamo come punto di partenza `example_cylinder`, uno degli esempi contenuti all'interno di LifeV nella cartella `example`. La versione originale di questo esempio permette di risolvere un problema di Navier–Stokes su una mesh cilindrica. Con semplici modifiche è possibile però utilizzare questo esempio su qualsiasi altra mesh.

Come tutti gli esempi e i test di LifeV, `example_cylinder` contiene un file di testo `data`, dove è possibile fissare il valore di alcuni parametri del problema, stabilire l'intervallo temporale della simulazione e scegliere gli spazi di elementi finiti, il tipo di precondizionatore e il solutore da utilizzare.

Analizziamo ora come avviene la lettura della mesh nella versione originale di LifeV. Il programma prende in input un file del tipo `.mesh`. La lettura di tale file viene svolta dall'Importer (file `ImporterMesh3D.hpp`). In particolare è la funzione `readINRIAMeshFile` che si occupa di leggere dal file punti, facce, volumi ed eventualmente lati della mesh, creare gli elementi geometrici e memorizzare il tutto all'interno di una classe `RegionMesh3D`

definita in `RegionMesh3D.hpp`. L'oggetto `RegionMesh3D`, oltre a contenere la lista dei vertici, dei lati, delle facce e dei volumi della mesh (tutti con il rispettivo marker e l'informazione se si tratta di elemento di bordo o interno), è in grado, tra le altre cose, di controllare ed eventualmente correggere la mesh che gli è stata fornita, costruire le facce interne e i lati mancanti nel file di input e costruire la mappa delle adiacenze tra le varie facce.

Una volta letta la mesh di partenza, essa viene poi partizionata tra i vari processori ad opera di `MeshPartitioner.hpp`. Questa operazione è utile per il calcolo in parallelo.

Dopo il partizionamento la mesh è utilizzata per la creazione degli spazi degli elementi finiti per la velocità e la pressione. Tali spazi ad elementi finiti sono memorizzati all'interno di una classe `FESpace` che contiene, oltre alla mesh partizionata, le informazioni relative al tipo di elemento finito che stiamo considerando e un puntatore alla lista dei gradi di libertà della mesh, che vengono generati dalla funzione `update` di `DOF.hpp`.

A questo punto nel file `cylinder.cpp` definiamo le condizioni al bordo che andremo ad utilizzare e che sono memorizzate all'interno di un oggetto `BCHandler`. Le condizioni al bordo sono definite dichiarando, oltre al tipo e alla funzione che andremo ad utilizzare, il marker delle facce a cui applicare la condizione.

A questo punto abbiamo tutte le informazioni per lanciare il solutore (nel caso in esame di tipo Oseen). Il programma fornisce in output, oltre alle informazioni sulla soluzione numerica e sui vari step del solutore, anche due file, `cylinder.h5` e `cylinder.xmf`, che possono essere letti da ParaView e permettono una visualizzazione grafica della soluzione numerica ottenuta. Tali file di output vengono generati per mezzo di un oggetto di tipo `Exporter`.

4.2 L'introduzione delle flag

La versione originale di LifeV prevedeva la distinzione solo tra elementi di bordo ed elementi interni. Sono però emerse da più parti diverse problematiche riguardo a questa limitazione: una è quella di nostro interesse, ovvero la possibilità di avere interfacce interne al dominio e la possibilità di imporre su di esse delle condizioni (cosa attualmente impossibile in LifeV); un'altra problematica è quella ad esempio relativa all'uso di elementi finiti di tipo XFem.

Di fronte all'evidenza dell'impossibilità, con gli attuali mezzi disponibili in LifeV, di affrontare questo tipo di problemi, si è deciso di porvi rimedio introducendo una flag che fornisse informazioni complete sulla natura dell'elemento all'interno della mesh che stiamo considerando.

Prima di entrare nel dettaglio, spieghiamo cosa è e come funziona una flag in LifeV.

Una flag è semplicemente un intero (senza segno) e si basa sulla rappresentazione binaria dei numeri. Ciò permette un utilizzo veloce ed intuitivo.

Supponiamo ad esempio che un oggetto possa avere tre caratteristiche A, B e C. Il primo passo è definire una flag primitiva per ciascuna di queste caratteristiche. Ciascuna flag primitiva deve corrispondere ad una potenza di 2. Ad esempio:

```
flag_Type UPDATE_A(1);
flag_Type UPDATE_B(2);
flag_Type UPDATE_C(4);
```

Usiamo potenze di 2 per rendere semplice e chiara la rappresentazione binaria: UPDATE_A corrisponde infatti a 001, UPDATE_B a 010, UPDATE_C a 100. In pratica il fatto che la caratteristica A sia attiva o meno è rappresentata da un 1 o uno 0 nel terzo bit; stessa cosa per B e il secondo bit e C e il primo bit. Se perciò vogliamo siano presenti sia A che C, dobbiamo avere valore 101, ovvero 5. Si ha

```
flag_Type UPDATE_AC(5);
```

Vediamo ora come è possibile combinare tra loro le flag. Supponiamo di voler combinare le flag UPDATE_A e UPDATE_AC. Se facciamo la somma, troviamo $1 + 5 = 6$, ovvero 110; non è quello che vogliamo fare. Per combinare tra loro le flag dobbiamo utilizzare gli operatori logici & (and) e | (or). Ad esempio $1|5 = 5$ (cioè $001|101 = 101$) che è esattamente ciò che vogliamo. L'operatore & ci permette invece di capire se una flag è accesa o meno. Ad esempio vogliamo sapere se la flag relativa ad A è accesa in UPDATE_AC: il comando UPDATE_A & UPDATE_AC ci fornisce 001; se A non fosse accesa ci fornirebbe 0.

Tornando al caso in esame e scegliendo come `flag_Type` un `Unsigned Int` a 32 bit, vediamo quali siano le flag introdotte (definizione nel file `MeshEntity.hpp`):

```
const flag_Type DEFAULT             ( 0x00 );
const flag_Type PHYSICAL_BOUNDARY   ( 0x01 );
const flag_Type INTERNAL_INTERFACE  ( 0x02 );
const flag_Type SUBDOMAIN_INTERFACE ( 0x04 );
const flag_Type OVERLAP             ( 0x08 );
const flag_Type CUTTED              ( 0x10 );
```

Notiamo anzitutto che i numeri sono scritti in base esadecimale, dunque il numero 10 in base 16 corrisponde a 16 in base decimale: si hanno perciò tutte potenze di 2, come descritto in precedenza.

La prima flag, DEFAULT, descrive il generico elemento interno e rappresenta il caso in cui nessuna flag è accesa, ovvero nessuna delle proprietà elencate è presente. La flag PHYSICAL_BOUNDARY rappresenta il caso in cui

l'elemento sia sul bordo fisico del dominio e corrisponde al vecchio booleano `boundary` usato in precedenza. La flag corrispondente al secondo numero, INTERNAL_INTERFACE, indica se l'elemento appartiene ad un'interfaccia interna al dominio che vogliamo mettere in evidenza e su cui potremo applicare condizioni al bordo di interfaccia. Le flag SUBDOMAIN_INTERFACE e OVERLAP sono utili per il calcolo parallelo e vogliono indicare gli elementi che si trovano al confine o nella zona di sovrapposizione tra una partizione e l'altra. Infine l'ultima flag si riferisce al caso particolare di elementi finiti tagliati.

Ovviamente un elemento della mesh può avere più flag accese contemporaneamente: un punto può ad esempio appartenere sia al bordo fisico che all'interfaccia, o una faccia può essere sia su un'interfaccia interna, sia sul bordo di un sottodominio di partizione.

Nel file `LifeV.hpp` sono definite alcune funzioni utili per poter utilizzare correttamente le flag:

```
namespace Flag
{
//! returns true if all byte-flags common set in refFlag
//  are also set in inputFlag
inline bool testAllSet ( flag_Type const & inputFlag,
                         flag_Type const & refFlag ) {
    return ( inputFlag  & refFlag ) == refFlag;
}

//! returns true if at least one flag set in refFlag
//    is set in inputFlag
inline bool testOneSet ( flag_Type const & inputFlag,
                         flag_Type const & refFlag ) {
    return inputFlag  & refFlag;
}

//! turns on the refFlag active bits in inputFlag
inline flag_Type turnOn  ( flag_Type const & inputFlag,
                           flag_Type const & refFlag ) {
    return inputFlag  | refFlag;
}

//! turns off the refFlag active bits in inputFlag
inline flag_Type turnOff ( flag_Type const & inputFlag,
                           flag_Type const & refFlag ) {
    return inputFlag  & ~refFlag;
}
```

```
//! switches the refFlag active bits in inputFlag
inline flag_Type change ( flag_Type const & inputFlag,
                          flag_Type const & refFlag ) {
    return inputFlag  ^ refFlag;
}

//! showMe method to print out flag status
//! the flag is converted to its binary form
//! ( right -> left corresponds to first -> last flag )
void showMe ( flag_Type const & flag, std::ostream & out );

//end namespace Flag
}
```

- `testAllSet` permette di confrontare tra loro due flag; ritorna `true` se tutte le flag accese in `refFlag` sono accese anche in `inputFlag`.
- `testOneSet` permette anch'essa il confronto tra due flag; ritorna `true` se almeno una flag accesa in `refFlag` è accesa anche in `inputFlag`.
- `turnOn` accende in `inputFlag` tutte le flag accese in `refFlag`.
- `turnOff` spegne in `inputFlag` tutte le flag accese in `refFlag`.
- `change` cambia in `inputFlag` il valore di tutte le flag accese in `refFlag`.
- `showMe` permette di ricevere come output il valore di una flag; esso è convertito in numero binario: la cifra più a destra corrisponde al valore della prima flag (PHYSICAL_BOUNDARY) e così via.

Il nostro lavoro per risolvere il problema delle interfacce è consistito in parte anche nell'adattare il codice di LifeV per l'utilizzo ottimale delle flag.

D'ora in avanti riportiamo tutte le modifiche principali che abbiamo dovuto introdurre a questo scopo nei file di LifeV contenuti nelle sottocartelle di `life`.

4.3 Lettura dei dati in input

4.3.1 Regole basilari per l'utilizzo del codice

Prima di descrivere in maniera approfondita tutte le modifiche da noi introdotte nella versione originale di LifeV è necessario introdurre alcune regole fondamentali che permettono di utilizzare al meglio il codice da noi scritto. Tali regole riguardano principalmente il tipo di file mesh da fornire in input al programma.

Anzitutto, benché LifeV accetti vari tipi di formati di file mesh, occorre che esso sia del tipo INRIA (file `.mesh`).

È inoltre fondamentale che le facce di interfaccia siano elencate insieme alle facce di bordo con un boundary marker che sia superiore a 1000000. La scelta di questo valore è dovuta al fatto che in LifeV esiste una classe `InternalEntitySelector` che permette di riconoscere elementi interni da quelli di bordo: essa definisce come interni tutti gli elementi con marker nullo o maggiore di 1000000. Dunque la scelta di un marker maggiore di 1000000 per le interfacce permette di mettere in evidenza alcune facce rispetto ad altre, pur facendole risultare comunque come interne.

È utile ma non necessario assegnare il boundary marker anche ai punti della mesh: il programma è in grado di fissare adeguatamente tali valori una volta noto il marker della faccia a cui appartengono. Notiamo che un punto, a differenza delle facce, può facilmente essere contemporaneamente di bordo e di interfaccia: in tale caso è opportuno assegnare a questi punti il marker relativo alla condizione al bordo che vogliamo sia effettivamente implementata in quel punto.

Infine, è importante tenere conto di alcune regole per l'assegnazione dei marker di volume. Il marker di volume per il nostro programma rappresenta il macrovolume a cui appartiene un certo elemento. È importante affinché il nostro codice funzioni che uno dei macrovolumi, ovvero le parti in cui il dominio è diviso per la presenza dell'interfaccia, abbia marker 1, mentre l'altro macrovolume abbia marker positivo e maggiore o uguale a 2.

Il nostro codice inoltre, nel caso di problemi senza condizioni di interfaccia, si comporta esattamente come nella versione originale di LifeV, senza che sia necessario apportare alcuna modifica ai dati in input.

4.3.2 MeshElement.hpp

Iniziamo ora a descrivere nel dettaglio le modifiche effettuate sul codice di LifeV. Partiamo da quelle relative al file `MeshElement.hpp`.

Questo file contiene una classe base per tutti gli elementi geometrici. Qui la modifica attuata è semplice e consiste nel cambiare l'ereditarietà pubblica della classe `MeshElement` da `MeshEntity` a `MeshEntityWithBoundary`. In questo modo tutti gli elementi geometrici avranno, oltre agli identificatori, una flag che ne indica il tipo e la posizione.

4.3.3 RegionMesh3D.hpp

Si tratta di un file che definisce una classe in grado di contenere tutte le informazioni relative ad una mesh. Qui le modifiche necessarie sono molte di più.

La prima è, come visto in precedenza, cambiare l'ereditarietà pubblica della classe da `MeshEntity` a `MeshEntityWithBoundary` per poter avere una flag per ogni elemento geometrico e poterne utilizzare le funzionalità.

Introduciamo nella classe `RegionMesh3D` alcuni attributi necessari per la gestione delle interfacce:

- `UInt M_numIFaces`: numero delle facce appartenenti all'interfaccia
- `UInt M_numMacroVolumes`: numero dei macrovolumi che compongono la mesh, ovvero le parti in cui il dominio è suddiviso per la presenza di interfacce. L'utilità di avere questa informazione sarà più chiara in seguito.

Ovviamente si è reso necessario introdurre anche semplici metodi per settare o ritornare il valore di questi attributi:

- `setNumIFaces`;
- `numIFaces` (equivalente a `numIElements`);
- `setNumMacroVolumes`
- `numMacroVolumes`

I metodi set fissano il valore dell'attributo a cui si riferiscono al numero intero messo tra parentesi, mentre gli altri metodi ritornano il valore dell'attributo.

Modifiche più consistenti sono state fatte invece all'interno della funzione `updateElementEdges`: tale funzione è quella che si occupa di creare la lista dei lati della mesh (se non sono forniti dalla mesh in input). Rispetto alla versione originale del codice abbiamo inserito una parte di codice che fosse in grado di settare accuratamente le flag dei lati in base al marker della faccia a cui appartengono. Infatti la parte originale del codice non era in grado di fissare correttamente il marker dei lati di interfaccia e riconoscerli come tali, poiché il controllo proposto nella versione originale è basato sul marker dei punti e non su quello delle facce: tale metodo è adatto per riconoscere lati di bordo (quelli con entrambi gli estremi di bordo) da quelli interni, ma non a riconoscere eventuali lati di interfaccia. Al contrario, siamo certi che lati appartenenti a facce di bordo sono anch'essi di bordo e quelli appartenenti a facce di interfaccia sono di interfaccia. Di conseguenza, è stato necessario modificare il ciclo della funzione, facendolo girare non sulle sole facce di bordo, ma su tutte le facce della mesh. Ecco le parti di modifiche appena descritte

```
if ( edgeList.empty() )
  {
    for ( typename Faces::iterator ifa = faceList.begin();
          ifa != faceList.end(); ++ifa )
```

```
{

  UInt mark = ifa->marker();
  for ( UInt j = 0; j < numLocalEdgesOfFace(); j++ )
  {
    i1 = bele.edgeToPoint( j, 0 );
    i2 = bele.edgeToPoint( j, 1 );
    i1 = ( ifa->point( i1 ) ).id();
    i2 = ( ifa->point( i2 ) ).id();

    _edge = makeBareEdge( i1, i2 );

    e = _be.addIfNotThere( _edge.first );

    if ( ce && e.second )
    {
    for ( UInt k = 0; k < 2 + FaceShape::S_numPointsPerEdge; k++ )
    {
       UInt inode = bele.edgeToPoint(j, k);
       edg.setPoint( k, ifa->point( inode ) );
     }
     edg.setMarker( mark );
     if (mark == 0)
     {
        edg.setFlag( Flag::turnOn ( edg.flag(), DEFAULT ) );
        addEdge (edg, false);
        edg.setFlag( Flag::turnOff ( edg.flag(), DEFAULT ) );
     }
     if ( (mark > 0) && (mark < 1000001) )
     {
       edg.setFlag( Flag::turnOn ( edg.flag(), PHYSICAL_BOUNDARY ) );
       addEdge (edg, true);
       edg.setFlag( Flag::turnOff ( edg.flag(), PHYSICAL_BOUNDARY ) );
       countBEdges++;
     }
     if (mark >= 1000001)
     {
       edg.setFlag( Flag::turnOn ( edg.flag(), INTERNAL_INTERFACE ) );
       addEdge (edg, false);
       edg.setFlag( Flag::turnOff ( edg.flag(), INTERNAL_INTERFACE ) );
     }
   }
 }
}
```

```
}
```

dove `countBEdges` è un contatore dei lati di bordo che useremo per fissare `M_numBEdges`.

Anche la funzione `updateElementFaces` ha richiesto profonde modifiche; essa infatti teneva in memoria nella versione originale solo le facce di bordo che erano fornite dalla mesh in input, poi cancellava le altre e le ricreava da capo come interne. In questo modo perdevamo però tutte le facce che sono fornite in input ma non sono di bordo, ovvero proprio le facce di interfaccia, che non erano più riconoscibili come tali dal programma. È stato dunque necessario introdurre un contenitore per le facce in input ma non di bordo, `_extraFaces`, e un booleano, `isExtra`. Come nella versione originale, il programma ora è in grado di memorizzare in `faceList` prima tutte le facce di bordo, poi quelle di interfaccia e in seguito quelle interne. Abbiamo infine fatto le necessarie modifiche per settare la flag e il marker di tutte le facce. Ecco la parte di codice di `updateElementFaces` in cui abbiamo apportato le modifiche descritte:

```
FaceType face;
MeshElementBareHandler<BareFace> _be;
MeshElementBareHandler<BareFace> _extraFaces;
std::pair<UInt, bool> e;
M_VToF.reshape( numLocalFaces(), numVolumes() );
UInt vid, i1, i2, i3, i4;
std::pair<BareFace, bool>_face;
GEOSHAPE ele;
if ( (faceList.size() == numFaces()) &
     getLinkSwitch( "FACES_HAVE_ADIACENCY" ) &
     getLinkSwitch( "HAS_ALL_FACES" ) )
{
    for ( typename Faces::iterator itf = faceList.begin();
          itf != faceList.end(); ++itf )
    {
        if ( itf->firstAdjacentElementPosition() != NotAnId
        && itf->firstAdjacentElementIdentity() != NotAnId)
            M_VToF( itf->firstAdjacentElementPosition() ,
            itf->firstAdjacentElementIdentity() ) = itf->localId();
            if ( itf->secondAdjacentElementPosition() != NotAnId
            && itf->secondAdjacentElementIdentity() != NotAnId)
                M_VToF( itf->secondAdjacentElementPosition(),
                 itf->secondAdjacentElementIdentity() ) = itf->localId();
    }
    setLinkSwitch( "HAS_VOLUME_TO_FACES" );
    if (verbose)
        std::cout << " done." << std::endl;
```

```
        return ;
}
UInt _numOriginalStoredFaces=faceList.size();
if ( ! faceList.empty() )
{
    std::pair<UInt, bool> _check;
    for ( UInt j = 0; j < faceList.size(); ++j )
    {
        i1 = ( faceList[ j ].point( 0 ) ).localId();
        i2 = ( faceList[ j ].point( 1 ) ).localId();
        i3 = ( faceList[ j ].point( 2 ) ).localId();
        if ( FaceShape::S_numVertices == 4 )
        {
            i4 = ( faceList[ j ].point( 3 ) ).localId();
            _face = makeBareFace( i1, i2, i3, i4 );
        }
        else
        {
            _face = makeBareFace( i1, i2, i3 );
        }
        _check = _be.addIfNotThere( _face.first );
        if (j>=this->M_numBFaces)
            _extraFaces.addIfNotThere( _face.first, j);
    }
}
for ( typename Volumes::iterator iv = volumeList.begin();
    iv != volumeList.end(); ++iv )
{
    vid = iv->localId();
    for ( UInt j = 0; j < numLocalFaces(); j++ )
    {
        i1 = ele.faceToPoint( j, 0 );
        i2 = ele.faceToPoint( j, 1 );
        i3 = ele.faceToPoint( j, 2 );
        i1 = ( iv->point( i1 ) ).localId();
        i2 = ( iv->point( i2 ) ).localId();
        i3 = ( iv->point( i3 ) ).localId();
        if ( FaceShape::S_numVertices == 4 )
        {
            i4 = ele.faceToPoint( j, 3 );
            i4 = ( iv->point( i4 ) ).localId();
            _face = makeBareFace( i1, i2, i3, i4 );
        }
        else
```

```
{
    _face = makeBareFace( i1, i2, i3 );
}
e = _be.addIfNotThere( _face.first );
M_VToF( j, vid ) = e.first;
bool _isBound=e.first<this->M_numBFaces;
bool _isExtra = (e.first >=this->M_numBFaces  &&
                 e.first < _numOriginalStoredFaces);
if (_isBound)
{
    FaceType & _thisFace(faceList[e.first]);
    _thisFace.firstAdjacentElementIdentity()   = vid;
    _thisFace.firstAdjacentElementPosition()   = j;
    _thisFace.secondAdjacentElementIdentity()  = NotAnId;
    _thisFace.secondAdjacentElementPosition()  = NotAnId;
}
else if (_isExtra)
{
    FaceType & _thisFace(faceList[e.first]);
    if(_extraFaces.deleteIfThere(_face.first))
    {
        for ( UInt k = 0; k < FaceType::S_numPoints; ++k )
        {
            _thisFace.setPoint(k,iv->point(ele.faceToPoint(j,k)));
        }
        _thisFace.firstAdjacentElementIdentity()  = vid;
        _thisFace.firstAdjacentElementPosition()  = j;
    }
    else
    {
        _thisFace.secondAdjacentElementIdentity()  = vid;
        _thisFace.secondAdjacentElementPosition()  = j;
    }
}
else if ( cf ) // A face not contained in the original list.
{
    if ( e.second )
    {
        for ( UInt k = 0; k < FaceType::S_numPoints; ++k )
            face.setPoint( k, iv->point( ele.faceToPoint( j, k ) ) );
        face.setFlag( Flag::turnOn ( face.flag(), DEFAULT ) );
        face.setMarker(0);
        face.firstAdjacentElementIdentity()  = vid;
        face.firstAdjacentElementPosition() = j;
```

```
                addFace( face, false );
            }
            else
            {
                faceList( e.first ).secondAdjacentElementIdentity() = vid;
                faceList( e.first ).secondAdjacentElementPosition() = j;
            }
        }
    }
}
```

4.3.4 ImporterMesh3D.hpp

È il file che permette di leggere un file di tipo `.mesh` e di memorizzare i dati ricavati in un oggetto `RegionMesh3D` definito in `RegionMesh3D.hpp`.

Qui le modifiche fatte sono numerose ed interessano sia l'introduzione delle flag sia la definizione di interfacce.

Una prima modifica basilare è quella di includere all'inizio anche il file `MeshEntity.hpp`, contenente le definizioni delle flag, in modo da poterle utilizzare.

La seconda modifica importante è quella relativa ai parametri in ingresso della funzione `readINRIAMeshFileHead`; a quelli già presenti abbiamo aggiunto il numero di vertici, di facce e di lati di interfaccia, in modo che tali valori possano essere poi settati al momento della chiamata.

La funzione su cui abbiamo concentrato la maggior parte delle nostre attenzioni è stata però `readINRIAMeshFile`. È la funzione che si occupa della lettura del file `.mesh` in input. Abbiamo dovuto modificarla per renderla in grado di settare in lettura le flag e trattare opportunamente gli elementi geometrici di interfaccia.

Per cominciare abbiamo introdotto tre struct `FiveNumbersInternal`, `FiveNumbersInterface` e `FiveNumbersVolume`, ciascuna con gli stessi membri e del tutto analoga alla struct `FiveNumbers` dell'originale. Per ciascuna abbiamo definito un vettore di tali strutture e il relativo iteratore.

In seguito, nella parte relativa alla lettura dei vertici abbiamo semplicemente aggiunto i comandi per il settaggio delle flag: i vertici con marker compreso tra 1 e 1000000 sono settati come PHYSICAL_BOUNDARY, quelli con marker 0 come DEFAULT, mentre quelli con marker maggiore o uguale a 1000001 sono INTERNAL_INTERFACE.

```
for ( i = 0; i < numberVertices; i++ )
{
    myStream >> x >> y >> z >> ibc;
    if ( !iSelect(entityFlag_Type(ibc))) //Boundary point
    {
```

```
        ++count;
        pointerPoint = &mesh.addPoint( true );
        pointerPoint->setMarker( entityFlag_Type( ibc ) );
        pointerPoint->setFlag( Flag::turnOn ( pointerPoint->flag(),
                                              PHYSICAL_BOUNDARY ) );
    }
    else
    {
        pointerPoint = &mesh.addPoint( false );
        pointerPoint->setFlag( Flag::turnOn ( pointerPoint->flag(),
                                              DEFAULT ) );
    }
    pointerPoint->setId     ( i );
    pointerPoint->setLocalId( i );
    pointerPoint->x() = x;
    pointerPoint->y() = y;
    pointerPoint->z() = z;
    pointerPoint->setMarker( entityFlag_Type( ibc ) );
    if (ibc > 1000000)
    {
        pointerPoint->setFlag( Flag::turnOn ( pointerPoint->flag(),
                                              INTERNAL_INTERFACE ) );
    }
    mesh.localToGlobalNode().insert( std::make_pair( i, i ) );
    mesh.globalToLocalNode().insert( std::make_pair( i, i ) );
}
```

Nella parte riservata alla lettura delle facce abbiamo deciso, nel caso le facce elencate nel file di input non siano solamente quelle di bordo, di memorizzare dapprima le facce di bordo, poi quelle di interfaccia e in seguito le eventuali facce interne memorizzate. Durante la lettura, se incontriamo una faccia di bordo (riconoscibile grazie alle solite convenzioni sul marker) la inseriamo direttamente in `pointerFace` con la flag PHYSICAL_BOUNDARY; se invece la faccia non è di bordo non la aggiungiamo alla lista delle facce della mesh, ma ne inseriamo i dati nel vettore `faceInterface` (se marker maggiore di 1000000) o `faceInternal` (se marker pari a 0). Poi inseriamo tutte le facce i cui dati sono contenuti in `faceInterface` in `faceList` come facce interne con flag INTERNAL_INTERFACE; nel contempo settiamo come INTERNAL_INTERFACE anche la flag dei punti che appartengono a tale faccia, in modo tale da avere fissate opportunamente tutte le flag dei vertici anche se non avessimo fornito abbastanza informazioni in input. Analogamente inseriamo poi le eventuali facce interne con flag DEFAULT. Ecco il codice con le modifiche:

```
UInt numberInterFaces =0;
```

```
for ( i = 0; i < numberStoredFaces; i++ )
{
    myStream >> p1 >> p2 >> p3 >> ibc;
    p1 -= idOffset; p2 -= idOffset; p3 -= idOffset;
    if ( numberStoredFaces > numberBoundaryFaces )
    {
        if (ibc!= 0 && ibc < 1000000) // boundary faces
        {
            pointerFace = &( mesh.addFace( true ) ); // Boundary faces
            pointerFace->setMarker( entityFlag_Type( ibc ) );
            pointerFace->setPoint( 0, mesh.point( p1 ) ); // set face conn.
            pointerFace->setPoint( 1, mesh.point( p2 ) ); // set face conn.
            pointerFace->setPoint( 2, mesh.point( p3 ) ); // set face conn.
            pointerFace->setBoundary( true);
            // Set as PHYSICAL_BOUNDARY not only face, but also vertices
            pointerFace->setFlag(Flag::turnOn(pointerFace->flag(),
                                              PHYSICAL_BOUNDARY ) );
            mesh.point(p1).setFlag(Flag::turnOn(mesh.point( p1 ).flag(),
                                                PHYSICAL_BOUNDARY ) );
            mesh.point(p2).setFlag(Flag::turnOn(mesh.point( p2 ).flag(),
                                                PHYSICAL_BOUNDARY ) );
            mesh.point(p3).setFlag(Flag::turnOn(mesh.point( p3 ).flag(),
                                                PHYSICAL_BOUNDARY ) );
        }
        if (ibc!= 0 && ibc > 1000000) // Interface face
        {
            faceInterfaceIterator->i1 = p1;
            faceInterfaceIterator->i2 = p2;
            faceInterfaceIterator->i3 = p3;
            faceInterfaceIterator->ibc = ibc;
            ++faceInterfaceIterator;
            numberInterFaces++;  // interface faces' counter
        }
        if (ibc==0)   // Internal face
        {
            faceInternalIterator->i1 = p1;
            faceInternalIterator->i2 = p2;
            faceInternalIterator->i3 = p3;
            faceInternalIterator->ibc = ibc;
            ++faceInternalIterator;
        }
    }
    else  // only boundary faces
    {
```

```
        pointerFace = &( mesh.addFace( true ) );
        pointerFace->setFlag( Flag::turnOn ( pointerFace->flag(),
                                             PHYSICAL_BOUNDARY ) );
        pointerFace->setMarker( entityFlag_Type( ibc ) );
        pointerFace->setPoint( 0, mesh.point( p1 ) ); // set face conn.
        pointerFace->setPoint( 1, mesh.point( p2 ) ); // set face conn.
        pointerFace->setPoint( 2, mesh.point( p3 ) ); // set face conn.
        pointerFace->setBoundary( true);
    }
}
faceInterface.resize( numberInterFaces);
faceInterfaceIterator = faceInterface.begin();
faceInternal.resize( numberStoredFaces - numberBoundaryFaces
                     - numberInterFaces);
faceInternalIterator = faceInternal.begin();
// Now I consider the faces in faceInterfaceIterator
for ( faceInterfaceIterator = faceInterface.begin();
      faceInterfaceIterator != faceInterface.end();
       ++faceInterfaceIterator )
{
    p1  = faceInterfaceIterator->i1;
    p2  = faceInterfaceIterator->i2;
    p3  = faceInterfaceIterator->i3;
    ibc = faceInterfaceIterator->ibc;
    pointerFace  = &( mesh.addFace( false ) );
    pointerFace->setMarker( entityFlag_Type( ibc ) );
    pointerFace->setPoint( 0, mesh.point( p1 ) ); // set face conn.
    pointerFace->setPoint( 1, mesh.point( p2 ) ); // set face conn.
    pointerFace->setPoint( 2, mesh.point( p3 ) ); // set face conn.
    pointerFace->setBoundary( false );
    // Set as INTERNAL_INTERFACE also vertices
    pointerFace->setFlag( Flag::turnOn ( pointerFace->flag(),
                                         INTERNAL_INTERFACE ) );
    mesh.point( p1 ).setFlag( Flag::turnOn ( mesh.point( p1 ).flag(),
                                             INTERNAL_INTERFACE ) );
    mesh.point( p2 ).setFlag( Flag::turnOn ( mesh.point( p2 ).flag(),
                                             INTERNAL_INTERFACE ) );
    mesh.point( p3 ).setFlag( Flag::turnOn ( mesh.point( p3 ).flag(),
                                             INTERNAL_INTERFACE ) );
}
// Now I consider the faces in faceInternal
for ( faceInternalIterator = faceInternal.begin();
      faceInternalIterator != faceInternal.end();
       ++faceInternalIterator )
```

```
{
    p1  = faceInternalIterator->i1;
    p2  = faceInternalIterator->i2;
    p3  = faceInternalIterator->i3;
    ibc = faceInternalIterator->ibc;
    pointerFace  = &( mesh.addFace( false ) ); // INTERNAL FACE
    pointerFace->setFlag( Flag::turnOn ( pointerFace->flag(), DEFAULT ) );
    pointerFace->setMarker( entityFlag_Type( ibc ) );
    pointerFace->setPoint( 0, mesh.point( p1 ) ); // set face conn.
    pointerFace->setPoint( 1, mesh.point( p2 ) ); // set face conn.
    pointerFace->setPoint( 2, mesh.point( p3 ) ); // set face conn.
    pointerFace->setBoundary( false);
}
```

Nella parte relativa ai lati, nella lettura dei lati di bordo abbiamo inserito il settaggio dell'opportuna flag.

Per quanto riguarda i volumi, infine, non ha senso parlare di interfaccia o bordo. Tuttavia abbiamo dovuto modificare anche qui il codice per fare in modo che venisse calcolato il numero di macrovolumi presenti nella mesh; vogliamo inoltre che nella lista dei volumi compaiano prima quelli appartenenti ad un macrovolume, poi quelli appartenenti all'altro; in questo modo facciamo sì che all'interfaccia l'orientazione delle facce sia sempre la stessa, cosicché, per come è costruito il codice, la normale di tale faccia è sempre uscente dallo stesso volume. Per farlo, durante la lettura dei dati da file vengono memorizzate tutte le informazioni relative ai volumi nel vettore `volume` e nel frattempo vengono contati i macrovolumi e inseriti in un vettore (`markerVector`) che poi viene riordinato. Per questioni che saranno chiare più avanti, è opportuno che il primo macrovolume abbia marker 1 e che gli altri abbiano marker positivo. In seguito inseriamo in `volumeList` tutti i tetraedri appartenenti al volume 1 memorizzati in `volume`, poi quelli con marker pari al secondo valore contenuto in `markerVector` e così via. Ecco le modifiche apportate al codice:

```
volume.resize( numberVolumes);
volumeIterator = volume.begin();
std::vector< UInt> markerVector;
UInt count2=0;
for ( i = 0; i < numberVolumes; i++ )
{
    myStream >> p1 >> p2 >> p3 >> p4 >> ibc;
    p1 -= idOffset; p2 -= idOffset; p3 -= idOffset; p4 -= idOffset;
    volumeIterator->i1 = p1;
    volumeIterator->i2 = p2;
    volumeIterator->i3 = p3;
```

```
    volumeIterator->i4 = p4;
    volumeIterator->ibc = ibc;
    ++volumeIterator;
    bool flagIs=false;
    for (UInt k=0; k < markerVector.size();k++)
    {
        if (ibc == markerVector[k])
        flagIs=true;
    }
    if (flagIs==false)
    {
        markerVector.push_back(ibc);
        count2++;
    }
}
sort(markerVector.begin(),markerVector.end());
numberMacroVolumes = markerVector.size();
mesh.setNumMacroVolumes( numberMacroVolumes);
count=0;
for (UInt k=0; k< markerVector.size();k++)
{
    for ( volumeIterator = volume.begin(); volumeIterator != volume.end();
          ++volumeIterator )
    {
        if (volumeIterator->ibc==markerVector[k])
        {
            pointerVolume = &mesh.addVolume();
            pointerVolume->setId     ( count );
            pointerVolume->setLocalId( count );
            pointerVolume->setPoint( 0, mesh.point( volumeIterator->i1) );
            pointerVolume->setPoint( 1, mesh.point( volumeIterator->i2 ) );
            pointerVolume->setPoint( 2, mesh.point( volumeIterator->i3 ) );
            pointerVolume->setPoint( 3, mesh.point( volumeIterator->i4 ) );
            pointerVolume->setMarker(entityFlag_Type(volumeIterator->ibc));
            count++;
         }
    }
}
```

4.3.5 ImporterMesh3D.cpp

È stato necessario anche modificare il file `ImporterMesh3D.cpp`. Oltre a modificare la chiamata di `readINRIAMeshFileHead` come descritto in prece-

denza, abbiamo fatto i cambiamenti necessari al codice per calcolare opportunamente il numero di vertici, lati e facce sull'interfaccia.

4.3.6 MeshPartitioner.hpp

Questo è il file che si occupa del partizionamento della mesh per il calcolo parallelo. Uno dei problemi che abbiamo incontrato con la versione originale di LifeV è il fatto che nella creazione delle partizioni venivano perse le informazioni relative alle flag e al marker per facce interne.

Nella classe `MeshPartitioner` abbiamo aggiunto l'attributo intero `M_numIFaces` che memorizzasse il numero di facce di interfaccia e i vettori `M_nInterfacePoints`, `M_nInterfaceEdges` e `M_nInterfaceFaces` che contenessero il numero di elementi di interfaccia per ogni partizione.

Abbiamo poi inserito all'interno dei metodi `constructNodes`, `constructEdges` e `constructFaces` i comandi necessari per contare il numero di elementi di interfaccia da inserire in tali vettori e accendere le flag di ciascun elemento.

```
template<typename MeshType>
void MeshPartitioner<MeshType>::constructNodes()
{
    UInt inode;
    for (UInt i = 0; i < M_numPartitions; ++i)
    {
        std::vector<Int>::iterator it;
        M_nBoundaryPoints[i] = 0;
        M_nInterfacePoints[i] = 0;
        (*M_meshPartitions)[i]->pointList.reserve(M_localNodes[i].size());
        ...
        for (it = M_localNodes[i].begin(); it != M_localNodes[i].end();
             ++it, ++inode)
        {
            typename MeshType::point_Type point = 0;
            bool boundary = M_originalMesh->isBoundaryPoint(*it);
            pp = &(*M_meshPartitions)[i]->addPoint(boundary);
            *pp = M_originalMesh->point( *it );
            pp->setLocalId( inode );
            pp->setFlag( Flag::turnOn ( pp->flag(),
                         M_originalMesh->point( *it ).flag() ) );
            if (boundary)
            {
                ++M_nBoundaryPoints[i];
            }
            if ( Flag::testOneSet(pp->flag(), INTERNAL_INTERFACE) )
```

```
            {
                ++M_nInterfacePoints[i];
            }
            (*M_meshPartitions)[i]->localToGlobalNode().insert(
                    std::make_pair( pp->localId(), pp->id() ));
            (*M_meshPartitions)[i]->globalToLocalNode().insert(
                    std::make_pair(pp->id(), pp->localId() ));
         }
    }
}

template<typename MeshType>
void MeshPartitioner<MeshType>::constructEdges()
{
    Int count;
    for (UInt i = 0; i < M_numPartitions; ++i)
    {
        ...
        for (is = M_localEdges[i].begin(); is != M_localEdges[i].end();
             ++is, ++count)
        {
            bool boundary = (M_originalMesh->isBoundaryEdge(*is));
            pe = &(*M_meshPartitions)[i]->addEdge(boundary);
            *pe = M_originalMesh->edge( *is );
            pe->setLocalId(count);
            pe->setFlag( Flag::turnOn ( pe->flag(),
                     M_originalMesh->edge( *is ).flag() ) );
            if (boundary)
            {
                ++M_nBoundaryEdges[i];
            }
            if ( Flag::testOneSet(pe->flag(), INTERNAL_INTERFACE) )
            {
                ++M_nInterfaceEdges[i];
            }
            ...
        }
    }
}

template<typename MeshType>
void MeshPartitioner<MeshType>::constructFaces()
{
    Int count;
```

```
    for (UInt i = 0; i < M_numPartitions; ++i)
    {
        std::map<Int, Int>::iterator im;
        std::set<Int>::iterator      is;
        typename MeshType::FaceType * pf = 0;
        UInt inode;
        count = 0;
        M_nBoundaryFaces[i] = 0;
        M_nInterfaceFaces[i] = 0;
        (*M_meshPartitions)[i]->faceList.reserve(
                M_localFaces[i].size());
        for (is = M_localFaces[i].begin(); is != M_localFaces[i].end();
             ++is, ++count)
        {
            bool boundary = (M_originalMesh->isBoundaryFace(*is));
            UInt flag = (M_originalMesh->face(*is).flag());
            if (Flag::testOneSet(flag, PHYSICAL_BOUNDARY))
            {
                ++M_nBoundaryFaces[i];
            }
            if ( Flag::testOneSet(flag, INTERNAL_INTERFACE) )
            {
                ++M_nInterFaces[i];
            }
            ...
       M_nInterfaceFaces[i] = M_nInterFaces[i];
    }
}
```

4.3.7 BCHandler.hpp

La classe `BCHandler` è una specie di contenitore per le condizioni al contorno. Quando si definisce una condizione al contorno deve essere dichiarato il marker delle facce su cui applicare tale condizione.

Il problema della versione originale di LifeV è il fatto che per applicare le condizioni viene fatto un ciclo sulle sole facce di bordo; ciò non permette di applicare eventuali condizioni su interfacce interne, come da noi desiderato.

Per ovviare a questo problema basta fare il ciclo non solo sulle facce di bordo, ma anche su quelle di interfaccia; il vantaggio è dato dal fatto che le facce di interfaccia, grazie alle modifiche viste in precedenza, sono memorizzate subito dopo quelle di bordo. Basta fare il seguente cambiamento:

```
numBElements = mesh.numBElements()+mesh.numIFaces();
```

```
//ORIGINALE numBElements = mesh.numBElements();
...
for ( ID iBoundaryElement = 0 ; iBoundaryElement < numBElements;
      ++iBoundaryElement )
```

4.4 Implementazione di uno spazio discontinuo

In questa sezione riportiamo tutte le ulteriori modifiche che abbiamo dovuto apportare al codice per poter creare ed utilizzare un nuovo spazio ad elementi finiti che fosse discontinuo all'interfaccia, cioè i cui gradi di libertà in corrispondenza dell'interfaccia fossero duplicati.

Anche in questo caso le modifiche interessano una grande quantità di file. Ciò è dovuto ad implemetazione non ottimale nella versione originale di LifeV, che nel costruire gli spazi ad elementi finiti utilizza non le informazioni relative ai gradi di libertà, ma quelle geometriche relative alla mesh di partenza.

4.4.1 DOF.hpp

Le modifiche più sostanziali sono quelle che interessano il file `DOF.hpp`. Questo file istanzia una classe `DOF` che si occupa di generare, a partire da una mesh e dal pattern locale dell'elemento finito, la lista dei gradi di libertà. Viene inoltre creata una mappa locale–globale che mette in relazione i gradi di libertà dell'elemento finito locale con la loro numerazione globale.

Il nostro interesse è a questo punto quello di poter raddoppiare tutti i gradi di libertà di interfaccia in modo da poter costruire uno spazio discontinuo. Ecco le modifiche necessarie.

Per prima cosa abbiamo dovuto includere all'inizio anche il file `MeshEntity.hpp` in modo da poter utilizzare tutte le funzionalità relative alle flag.

Abbiamo poi introdotto tra gli attributi privati della classe `DOF` gli interi `M_doubleDof`, `M_doubleDofOnVertices`, `M_doubleDofOnEdges` e `M_doubleDofOnFaces`, indicanti il numero totale dei DOF ripetuti e quello relativo ai soli DOF ripetuti su vertici, lati e facce. Ovviamente sono stati aggiunti anche i metodi necessari per settare e ottenere questi attributi.

Tra i membri pubblici invece sono stati introdotti quattro vettori del tipo `std::vector < std::pair<ID,ID> >`, chiamati `M_doubledDofList`, `M_doubledDofOnVerticesList`, `M_doubledDofOnEdgesList` e `M_doubledDofOnFacesList`, contenenti le liste di coppie formate dal grado di libertà originale e dal suo corrispondente duplicato.

La funzione `update` è quella che si occupa di generare i gradi di libertà e le mappe; è qui che si sono concentrati tutti i cambiamenti.

Per prima cosa abbiamo dovuto distinguere tra `M_globalDof` e `M_totalDof`: il primo contiene il numero dei gradi di libertà che avremmo nel caso clas-

sico senza interfacce, mentre il secondo contiene l'effettivo numero totale dei gradi di libertà, compresi quelli ripetuti. Ovviamente, se non esistono interfacce i due valori saranno coincidenti fino alla fine.

Nel ciclo che si occupa di generare i DOF sui vertici abbiamo introdotto le modifiche necessarie per la duplicazione dei DOF. Se un vertice non è di interfaccia, il codice si comporta esattamente come nella versione originale: viene calcolato il label, ovvero l'ID del punto della mesh, e inserito all'interno della mappa `M_localToGlobal`. Se invece è di interfaccia, si deve controllarne il macrovolume di appartenenza: se l'elemento a cui appartiene fa parte del macrovolume con marker 1, ci si comporta come nel caso precedente. Se invece l'elemento appartiene ad un volume con marker diverso da 1, allora deve essere duplicato. Per prima cosa controlliamo all'interno della lista `M_doubledDofOnverticesList` se abbiamo già duplicato o meno il vertice in esame. Nel caso in cui esso non sia ancora stato duplicato, dobbiamo assegnare un label, ovvero un ID, in modo che i DOF duplicati sui vertici seguano come numerazione i DOF non duplicati. Assegniamo il nuovo label alla mappa locale–globale e incrementiamo il valore di `M_doubledDof`, `M_doubledDofOnVertices` e `M_totalDof`; inoltre aggiungiamo la coppia DOF–originale/DOF–duplicato alla lista `M_doubledDofOnVerticesList`.

Lo stesso procedimento deve essere ripetuto per i lati, facendo in modo che la numerazione dei DOF riparta da quella relativa all'ultimo grado di libertà di vertice ripetuto inserito. La stessa cosa deve accadere per i gradi di libertà sulla faccia, mentre non è necessario apportare alcuna modifica per i gradi di libertà di volume, dato che non ha senso parlare di DOF di volume di interfaccia.

Ecco la funzione update con tutte le modifiche:

```
template <typename MeshType>
void DOF::update( MeshType& mesh )
{
  typedef typename MeshType::ElementShape GeoShapeType;
  UInt nbLocalDofPerEdge = M_elementDofPattern.nbDofPerEdge();
  UInt nbLocalDofPerVertex = M_elementDofPattern.nbDofPerVertex();
  UInt nbLocalDofPerFace = M_elementDofPattern.nbDofPerFace();
  UInt nbLocalDofPerVolume = M_elementDofPattern.nbDofPerVolume();
  M_nbLocalVertex = GeoShapeType::S_numVertices;
  M_nbLocalEdge = GeoShapeType::S_numEdges;
  M_nbLocalFace = GeoShapeType::S_numFaces;
  M_numElement = mesh.numElements();
  UInt nbGlobalVolume = mesh.numGlobalVolumes();
  UInt nbGlobalEdge = mesh.numGlobalEdges();
  UInt nbGlobalVertex = mesh.numGlobalVertices();
  UInt nbGlobalFace = mesh.numGlobalFaces();
```

```
  UInt i, l, ie;
  // Number of macrovolumes in the mesh
  UInt nbMacroVolumes = mesh.numMacroVolumes();
  // Total number of degree of freedom for each element
  UInt nldof = nbLocalDofPerVolume
      + nbLocalDofPerEdge * M_nbLocalEdge
      + nbLocalDofPerVertex * M_nbLocalVertex
      + nbLocalDofPerFace * M_nbLocalFace;
  // Consistency check
  ASSERT_PRE( nldof == UInt( M_elementDofPattern.nbLocalDof() ),
              "Something wrong in FE specification" ) ;
// Global total of degrees of freedom; it's different from M_totalDof,
// which is the actual total of DOFs
// and takes into account also repeated DOFs
  UInt M_globalDof = nbGlobalVolume * nbLocalDofPerVolume
      + nbGlobalEdge * nbLocalDofPerEdge
      + nbGlobalVertex * nbLocalDofPerVertex
      + nbGlobalFace * nbLocalDofPerFace;
// If there are no doubled DOFs, M_totalDof = M_globalDof till the end
  M_totalDof = M_globalDof;
  // Reshape the container to fit the needs
  M_localToGlobal.reshape( nldof, M_numElement );
// Make sure the mesh has everything needed
  bool update_edges( nbLocalDofPerEdge != 0 && ! mesh.hasLocalEdges() );
  if ( update_edges )
  {
    mesh.updateElementEdges();
  }
  bool update_faces( nbLocalDofPerFace != 0 && ! mesh.hasLocalFaces() );
  if ( update_faces )
  {
    mesh.updateElementFaces();
  }
  UInt gcount( 0 );
  UInt lcount;
  UInt lc;
  std::pair<ID,ID> couple;
  UInt kk=0;
  // Vertex Based Dof
  M_dofPositionByEntity[ 0 ] = gcount;
  if ( nbLocalDofPerVertex > 0 )
    for ( ie = 0; ie < M_numElement; ++ie )//for each element
    {
      lc = 0;
```

```
      for ( i = 0; i < M_nbLocalVertex; ++i )//for each vertex
      {
 // label of the ith point of the mesh element
        UInt dofLabel = gcount + mesh.element( ie ).point( i ).id();
// label of the corresponding doubled DOF; if 0, the DOF will not be repeated
        UInt doubledDofLabel = 0;
// If the DOF is not an interface point and/or belongs to macrovolume 1,
// has label dofLabel, else it has to be doubled
        M_localToGlobal( lc++, ie ) = dofLabel;
// If the DOF must be repeated
        if ( (nbMacroVolumes > 1)  && (mesh.element( ie ).marker() > 1) &&
             ( Flag::testOneSet( mesh.element( ie ).point( i ).flag(),
             INTERNAL_INTERFACE )))
        {
// Is already in M_doubledDofOnVerticesList?
          bool isAlready = false;
          for (UInt k=0; k < M_doubledDofOnVerticesList.size(); k++)
          {
            if (dofLabel == M_doubledDofOnVerticesList[k].first)
            {
//If the DOF with ID = dofLabel is already in M_doubledDofOnVerticesList...
              M_localToGlobal( lc-1, ie ) =
                  M_doubledDofOnVerticesList[k].second;
              isAlready = true;
              break;
            }
          }
//If the DOF with ID = dofLabel is not yet in M_doubledDofOnVerticesList,
// it must be added
          if ( !isAlready )
          {
            doubledDofLabel = nbGlobalVertex+kk;
            M_localToGlobal( lc-1, ie ) = doubledDofLabel;
            couple = std::make_pair ( dofLabel, doubledDofLabel);
            M_doubledDofOnVerticesList.push_back( couple );
            M_doubledDofList.push_back( couple );
            kk++;
            M_totalDof++;
          }
        }
      }
    }
  M_doubledDofOnVertices=M_doubledDofOnVerticesList.size();
  // End Vertex Based Dof
```

```
  gcount += nbGlobalVertex*nbLocalDofPerVertex + kk;
  lcount = nbLocalDofPerVertex * M_nbLocalVertex;
  M_dofPositionByEntity[ 1 ] = gcount;

  // Edge based Dof
  kk = 0;
  if ( nbLocalDofPerEdge > 0 )
    for ( ie = 0; ie < M_numElement; ++ie )//for each element
    {
      lc = lcount;
      for ( i = 0; i < M_nbLocalEdge; ++i )//for each edge in the element
        for ( l = 0; l < nbLocalDofPerEdge; ++l )//for each dof per edge
        {
// label of the ith point of the mesh element
          UInt eID = mesh.edgeList(mesh.localEdgeId(ie, i)).id();
          UInt dofLabel = gcount + eID * nbLocalDofPerEdge + l;
          // label of doubled Dof
          UInt doubledDofLabel = 0;
          M_localToGlobal( lc++, ie ) = dofLabel;
          if ( ( nbMacroVolumes > 1 ) &&
               ( Flag::testOneSet(mesh.edgeList(
                     mesh.localEdgeId(ie, i)).flag(),
                     INTERNAL_INTERFACE ) ) &&
             ( mesh.element( ie ).marker() > 1 ) )
          {
// Is already in M_doubledDofOnEdgesList?
            bool isAlready = false;
            for (UInt k=0; k < M_doubledDofOnEdgesList.size(); k++)
            {
              if (dofLabel == M_doubledDofOnEdgesList[k].first)
              {
//If dofLabel is already in M_doubledDofOnEdgesList...
                isAlready = true;
                M_localToGlobal( lc-1, ie ) =
                    M_doubledDofOnEdgesList[k].second;
                break;
              }
            }
            if ( !isAlready )
            {
              doubledDofLabel = nbGlobalEdge + kk;
              M_localToGlobal( lc-1, ie ) = doubledDofLabel;
              couple = std::make_pair ( dofLabel, doubledDofLabel);
              M_doubledDofOnEdgesList.push_back( couple );
```

```
              M_doubledDofList.push_back( couple );
              kk++;
              M_totalDof++;
             }
          }
        }
    }
  // End Edge Based Dof
  // Face Based Dof
  gcount += nbGlobalEdge * nbLocalDofPerEdge + kk;
  lcount += nbLocalDofPerEdge * M_nbLocalEdge;
  M_dofPositionByEntity[ 2 ] = gcount;
  kk = 0;
  if ( nbLocalDofPerFace > 0 )
    for ( ie = 0; ie < M_numElement; ++ie )//for each element
    {
      lc = lcount;
#ifdef TWODIM
//In 2D there are no faces of interfaces
// when working in 2D we simply iterate over the elements to have faces
      for ( l = 0; l < nbLocalDofPerFace; ++l )
        M_localToGlobal( lc++, ie ) =
            gcount + ( ie  ) * nbLocalDofPerFace + l;
#else // THREEDIM
      for ( i = 0; i < M_nbLocalFace; ++i )//for each face in the element
        for ( l = 0; l < nbLocalDofPerFace; ++l )//for each dof per face
        {
// label of the ith point of the mesh element
          UInt dofLabel = gcount +
               mesh.element( ie ).point( i ).id() * nbLocalDofPerEdge + l;
          UInt doubledDofLabel = 0;
          M_localToGlobal( lc++, ie ) = dofLabel;
          if ( ( nbMacroVolumes > 1 ) &&
               ( Flag::testOneSet( mesh.faceList( mesh.localFaceId
                 ( ie, i ) ).flag(), INTERNAL_INTERFACE ) ) &&
               ( mesh.element( ie ).marker() > 1 ) )
          {
// Is already in M_doubledDofOnFacesList?
            bool isAlready = false;
            for (UInt k=0; k < M_doubledDofOnFacesList.size(); k++)
            {
              if (dofLabel == M_doubledDofOnFacesList[k].first)
              {
//If dofLabel is already in M_doubledDofOnFacesList...
```

```
                isAlready = true;
                M_localToGlobal( lc-1, ie ) =
                   M_doubledDofOnFacesList[k].second;
                break;
              }
            }
            if ( !isAlready )
            {
              doubledDofLabel = nbGlobalFace + kk;
              M_localToGlobal( lc-1, ie ) = doubledDofLabel;
              couple = std::make_pair ( dofLabel, doubledDofLabel);
              M_doubledDofOnFacesList.push_back( couple );
              M_doubledDofList.push_back( couple );
              kk++;
              M_totalDof++;
            }
          }
        }
#endif
  }
  // End Face Based Dof
  // Volume  Based Dof
  gcount += nbGlobalFace * nbLocalDofPerFace + kk;
  lcount += nbLocalDofPerFace * M_nbLocalFace;
  M_dofPositionByEntity[ 3 ] = gcount;
// There are no volume DOF on the interface, we don't need to double them
  if ( nbLocalDofPerVolume > 0 )
    for ( ie = 0; ie < M_numElement; ++ie )
    {
      lc = lcount;
      for ( l = 0; l < nbLocalDofPerVolume; ++l )
      {
        M_localToGlobal( lc++, ie ) = gcount +
           mesh.element( ie ).id()* nbLocalDofPerVolume + l;
      }
    }
  gcount += nbGlobalVolume * nbLocalDofPerVolume;
  M_dofPositionByEntity[ 4 ] = gcount;
...
}
```

4.4.2 DOF.cpp

Nel file `DOF.cpp` è stato modificato il costruttore di copia, in modo che tenesse conto anche di tutte le variabili introdotte per gestire i DOF duplicati.

```
DOF::DOF( const DOF & dof2 ) :
M_elementDofPattern( dof2.M_elementDofPattern ), //,
    M_offset( dof2.M_offset ),
M_totalDof( dof2.M_totalDof ),
M_doubledDof( dof2.M_doubledDof ),
M_doubledDofOnVertices( dof2.M_doubledDofOnVertices ),
M_doubledDofOnEdges( dof2.M_doubledDofOnEdges ),
M_doubledDofOnFaces( dof2.M_doubledDofOnFaces ),
M_doubledDofOnVolumes( dof2.M_doubledDofOnVolumes ),
M_doubledDofList( dof2.M_doubledDofList ),
M_doubledDofOnVerticesList( dof2.M_doubledDofOnVerticesList ),
M_doubledDofOnEdgesList( dof2.M_doubledDofOnEdgesList ),
M_doubledDofOnFacesList( dof2.M_doubledDofOnFacesList ),
M_doubledDofOnVolumesList( dof2.M_doubledDofOnVolumesList ),
M_numElement( dof2.M_numElement ),
M_nbLocalVertex( dof2.M_nbLocalVertex ),
    M_nbLocalEdge( dof2.M_nbLocalEdge ),
    M_nbLocalFace( dof2.M_nbLocalFace ),
M_localToGlobal( dof2.M_localToGlobal ),
    M_nbFace(dof2.M_nbFace),
M_localToGlobalByFace(dof2.M_localToGlobalByFace),
    M_globalToLocalByFace(dof2.M_globalToLocalByFace),
M_faceToPoint(dof2.M_faceToPoint),
    M_numLocalDofByFace(dof2.M_numLocalDofByFace)
{
if ( &dof2 == this )
return ;

for ( UInt i = 0; i < 5; ++i )
M_dofPositionByEntity[ i ] =
             dof2.M_dofPositionByEntity[ i ];
}
```

4.4.3 FESpace.hpp

In questo file è implementata una classe `FESpace` contenente tutte le informazioni necessarie per costruire uno spazio di elementi finiti. Anche in questo caso la gestione di DOF ripetuti richiede alcune modifiche. In particolare è stata introdotta una funzione `createMap`, che serve a costruire la

mappa dei gradi di libertà. Tale funzione serve a correggere una precedente implementazione non ottimale di LifeV, che nel costruire lo spazio ad elementi finiti utilizzava le informazioni geometriche della mesh e non quelle relative ai gradi di libertà, rischiando, come nel nostro caso, di perdere informazioni.

La funzione `createMap` costruisce l'insieme `dofNumberSet` dei DOF (memorizzati per mezzo della loro numerazione globale). Usiamo un set per evitare ripetizioni. Gli elementi contenuti nel set vengono poi copiati all'interno del vettore `myGlobalElements`, che sarà poi utile per utilizzare il costruttore di `MapEpetra` che vedremo in seguito. Utilizzando tale vettore creiamo la mappa dei DOF, memorizzata in `M_map`.

```
template<typename MeshType, typename MapType>
void
FESpace<MeshType,MapType>::
createMap(const commPtr_Type& commptr)
{
    std::set<Int> dofNumberSet;
// Gather all dofs local to the given mesh (dofs use global numbering)
// The set ensures no repetition
    for (UInt elementId=0; elementId < this->M_mesh->numElements();
          ++elementId )
        for (UInt localDof=0; localDof < this->M_dof->numLocalDof();
              ++localDof )
        {
            dofNumberSet.insert( static_cast<Int>( this->M_dof
                ->localToGlobalMap(elementId,localDof ) ) );
        }
// dump the set into a vector for adjacency
// to save memory I use copy() and not the vector constructor directly
    std::vector<Int> myGlobalElements(dofNumberSet.size());
    std::copy(dofNumberSet.begin(),dofNumberSet.end(),
              myGlobalElements.begin());
// Save memory
    dofNumberSet.clear();
// Create the map
    MapType map( -1,myGlobalElements.size(),
                 &myGlobalElements[0],commptr );
// Store the map. If more than one field is present the map is
// duplicated by offsetting the DOFs
    for ( UInt ii = 0; ii < M_fieldDim; ++ii )
        *M_map += map;
}
```

4.4.4 MapEpetra.hpp

La classe `MapEpetra` fornisce un'interfaccia per LifeV della classe `Epetra_Map` contenuta in Trilinos. Questa si occupa di gestire la distribuzione degli elementi di matrici e vettori tra i vari processori per il calcolo in parallelo.

Per i nostri scopi abbiamo dovuto includere il file `DOF.hpp` e modificare la classe `MapEpetra` inserendo un costruttore che prendesse in ingresso un elemento `DOF` per riuscire a passare anche l'elenco dei gradi di libertà ripetuti.

Questo costruttore utilizza non solo i vertici, lati e facce della mesh, ma anche gli eventuali gradi di libertà ripetuti per costruire i vettori `repeatedNodeVector`, `repeatedEdgeVector` e `repeatedFaceVector`. Questi vettori sono poi dati in input alla funzione `setUp`.

```
template<typename Mesh>
MapEpetra::
MapEpetra( const ReferenceFE&  refFE,
           const Mesh&         mesh,
           const DOF&          dof,
           const comm_ptrtype& commPtr ):
           M_repeatedMapEpetra(),
           M_uniqueMapEpetra(),
           M_exporter(),
           M_importer(),
           M_commPtr( commPtr )
{
    std::vector<Int> repeatedNodeVector;
    std::vector<Int> repeatedEdgeVector;
    std::vector<Int> repeatedFaceVector;
    std::vector<Int> repeatedVolumeVector;
    if ( refFE.nbDofPerVertex() )
    {
        repeatedNodeVector.reserve(mesh.numPoints()+
             dof.numDoubledDofOnVertices());
        for ( UInt ii = 0; ii < mesh.numPoints(); ii++ )
        {
            repeatedNodeVector.push_back(mesh.pointList(ii).id() );
        }
        for ( UInt ii = 0; ii < dof.numDoubledDofOnVertices(); ii++ )
        {
            repeatedNodeVector.push_back(
               dof.M_doubledDofOnVerticesList[ii].second);
        }
    }
    if ( refFE.nbDofPerEdge() )
    {
```

```
        repeatedEdgeVector.reserve( mesh.numEdges() );
        for ( UInt ii = 0; ii < mesh.numEdges(); ii++ )
            repeatedEdgeVector.push_back( mesh.edgeList(ii).id() );
    }
    if ( refFE.nbDofPerFace() )
    {
        repeatedFaceVector.reserve( mesh.numFaces() );
        for ( UInt ii = 0; ii < mesh.numFaces(); ii++ )
            repeatedFaceVector.push_back( mesh.faceList(ii).id() );
    }
    if ( refFE.nbDofPerVolume() )
    {
        repeatedVolumeVector.reserve( mesh.numVolumes() );
        for ( UInt ii = 0; ii < mesh.numVolumes(); ii++ )
            repeatedVolumeVector.push_back( mesh.volumeList(ii).id() );
    }
    setUp( refFE,
           commPtr,
           repeatedNodeVector,
           repeatedEdgeVector,
           repeatedFaceVector,
           repeatedVolumeVector );
}
```

4.5 Il solutore

4.5.1 OseenSolver.hpp

Questo file contiene un solutore per equazioni di Oseen. Contiene informazioni sulla mesh, sul tipo di solutore, sulle matrici del sistema e sul precondizionatore e fornisce la soluzione del problema. Il sistema lineare risultante viene risolto per mezzo del metodo GMRES sulla matrice completa, contenente sia le informazioni relative alla velocità sia quelle relative alla pressione.

Nel solutore abbiamo dovuto agire sull'imposizione della continuità per la velocità, imponendo condizioni di salto nullo per ogni componente per ogni coppia di gradi di libertà ripetuti. In pratica, abbiamo implementato la teoria introdotta nella sezione 3.6.

Riportiamo le modifiche più importanti al codice: tali modifiche si riferiscono al metodo `applyBoundaryCondition`, che come è chiaro, interviene in fase di applicazione delle condizioni al bordo alle matrici del sistema algebrico.

```
if (M_velocityFESpace.dof().M_doubledDofOnVerticesList.size()!=0)
```

```
{
//! Z matrix
matrixPtr_Type                M_matrixZ;
matrixPtr_Type                M_matrixK;
matrixPtr_Type                M_matrixA;
matrixPtr_Type                M_matrixTMP;
matrixPtr_Type                M_matrixEND;
vector_Type M_vectorB( rightHandSide, Unique );
M_matrixZ.reset  ( new matrix_Type( M_localMap ) );
M_matrixK.reset  ( new matrix_Type( M_localMap ) );
M_matrixA.reset  ( new matrix_Type( M_localMap ) );
M_matrixTMP.reset  ( new matrix_Type( M_localMap ) );
M_matrixEND.reset  ( new matrix_Type( M_localMap ) );
double K=5.;    //Lagrange Multiplier
M_matrixZ->insertOneDiagonal();
for (UInt k1=0;
     k1 < M_velocityFESpace.dof().M_doubledDofOnVerticesList.size();
     k1++)
{
    // Loop on components invoved in this boundary condition
    for ( ID j = 0; j < 3; ++j )
    {
    UInt originalDof =
       M_velocityFESpace.dof().M_doubledDofOnVerticesList[k1].first
       + j * M_velocityFESpace.dof().numTotalDof();
    UInt doubledDof  =
       M_velocityFESpace.dof().M_doubledDofOnVerticesList[k1].second
       + j * M_velocityFESpace.dof().numTotalDof();
    M_matrixZ->addToCoefficient( originalDof, originalDof, -0.5 );
    M_matrixZ->addToCoefficient( originalDof, doubledDof,  0.5 );
    M_matrixZ->addToCoefficient( doubledDof,  originalDof, 0.5 );
    M_matrixZ->addToCoefficient( doubledDof,  doubledDof   -0.5 );

    M_matrixK->addToCoefficient(  originalDof, originalDof, K );
    M_matrixK->addToCoefficient( originalDof,  doubledDof,  -K );
    M_matrixK->addToCoefficient( doubledDof,   originalDof, -K );
    M_matrixK->addToCoefficient( doubledDof,   doubledDof,  K );
    }
}
M_matrixZ->globalAssemble();
M_matrixK->globalAssemble();
M_matrixZ->multiply(false, matrix, false, *M_matrixTMP, true);
M_matrixTMP->multiply(false, *M_matrixZ, false, *M_matrixEND, true);
matrix = *M_matrixEND;
```

```
    matrix += *M_matrixK;
    matrix.globalAssemble();
    M_matrixZ->multiply(false, rightHandSideFull, M_vectorB);
    rightHandSideFull = M_vectorB;
    }
rightHandSide = rightHandSideFull;
...
} // applyBoundaryCondition
```

4.6 Output dei risultati

4.6.1 Exporter.hpp

Il file `Exporter.hpp` istanzia classi e funzionalità che sono utili per il post–processing dei dati. Anche qui occorrono modifiche per fare in modo che l'esportazione dei dati tenga conto anche dei gradi di libertà ripetuti.

Per utilizzare le liste dei gradi di libertà abbiamo dovuto includere il file `DOF.hpp` e inserire nella classe `Exporter` due membri di tipo `M_velocityDof` e `M_pressureDof`.

Allo stesso tempo abbiamo modificato il costruttore in modo tale che prendesse in input anche due oggetti di tipo `FESpace` corrispondenti agli spazi di elementi finiti di velocità e pressione.

```
template<typename MeshType>
Exporter<MeshType>::Exporter(
       const GetPot& dfile, const std::string& prefix,
       const FESpace<mesh_Type, MapEpetra>& velocityFESpace,
       const FESpace<mesh_Type, MapEpetra>& pressureFESpace  ):
    M_prefix        ( prefix ),
    M_postDir       ( dfile("exporter/post_dir", "./") ),
    M_timeIndexStart( dfile("exporter/start",0) ),
    M_timeIndex     ( M_timeIndexStart ),
    M_velocityDof (velocityFESpace.dof() ),
    M_pressureDof (pressureFESpace.dof() ),
    M_save          ( dfile("exporter/save",1) ),
    M_multimesh     ( dfile("exporter/multimesh",true) ),
    M_timeIndexWidth( dfile("exporter/time_id_width",5) )
{}
```

4.6.2 ExporterHDF5.hpp

La classe `ExporterHDF5` è una particolare istanza di `Exporter`, da cui deriva pubblicamente, che serve ad esportare il valore di una certa variabile in formato HDF5 in modo tale da permetterne la visualizzazione grafica tramite ParaView.

La versione originale di questo exporter non ci permette di ottenere una visualizzazione dei risultati del nostro problema con interfaccia in quanto, basandosi sui punti anziché sui DOF, non riesce a trattare il caso dei nodi di interfaccia.

Si sono rivelate necessarie dunque diverse modifiche. In primis l'aggiunta dei due spazi di elementi finiti di velocità e pressione al costruttore, in modo da poter utilizzare il costruttore di `Exporter` definito in precedenza.

Abbiamo inoltre sostituito il metodo `writeGeometry` con il metodo `writeGeometry_mod`. In tale versione modificata per costruire il vettore `connections` utilizziamo non gli ID dei punti, ma quelli dei DOF, ottenuti per mezzo della `localToGlobalMap` costruita in precedenza e memorizzata in `M_velocityDof` e in `M_pressureDof`. Usiamo inoltre il costruttore di `MapEpetra` con argomento di tipo `DOF` per creare la mappa `tmpMapP1`. Per costruire i vettori `pointsX`, `pointsY` e `pointsZ` con le coordinate dei punti, non cicliamo solo sui vertici della mesh, ma cicliamo sui DOF, compresi quelli ripetuti.

```
template <typename MeshType>
void ExporterHDF5<MeshType>::writeGeometry_mod()
{
    UInt numberOfPoints = MeshType::ElementShape::S_numPoints;
    std::vector<Int> elementList;
    elementList.reserve(this->M_mesh->numElements()*numberOfPoints);
    for (ID i=0; i < this->M_mesh->numElements(); ++i)
    {
        typename MeshType::ElementType const& element
                              (this->M_mesh->element(i));
        UInt lid= i*numberOfPoints;
        for (ID j=0; j< numberOfPoints; ++j, ++lid)
        {
            elementList[lid] = element.id()*numberOfPoints+j;
        }
    }
    Epetra_Map connectionsMap(
         this->M_mesh->numGlobalElements()*numberOfPoints,
         this->M_mesh->numElements()*numberOfPoints,
         &elementList[0],
         0,
         this->M_dataVector.begin()->storedArrayPtr()->comm() );
    Epetra_IntVector connections(connectionsMap);
    for (ID i=0; i < this->M_mesh->numElements(); ++i)
    {
        UInt lid=i*numberOfPoints;
        for (ID j=0; j< numberOfPoints; ++j, ++lid)
```

```
        {
            connections[lid] =
                this->M_velocityDof.localToGlobalMap(i,j);
        }
    }
    this->M_dataVector.begin()->storedArrayPtr()->comm().Barrier();
    // Points
// Build a map for linear elements, even though the original FE might be P0
// This gives the right map for the coordinate arrays
    MapEpetra subMap;
    switch ( MeshType::ElementShape::S_shape )
    {
    case TETRA:
    {
        const ReferenceFE & refFEP1 = feTetraP1;
        MapEpetra tmpMapP1(
             refFEP1,
             *this->M_mesh,
             this->M_velocityDof,
             this->M_dataVector.begin()->storedArrayPtr()
                    ->mapPtr()->commPtr());
        subMap = tmpMapP1;
        break;
    }
    ...
    }
    VectorEpetra pointsX(subMap);
    VectorEpetra pointsY(subMap);
    VectorEpetra pointsZ(subMap);
    Int gid;
    //Cycle for nodes
    for (ID i=0; i < this->M_mesh->numVertices(); ++i)
    {
// Saving the initial mesh if M_multimesh is false (important for restart)
        typename MeshType::point_Type point;
        if ( this->M_multimesh )
            point = this->M_mesh->point(i);
        else
            point = this->M_mesh->pointInitial(i);
        gid = point.id();
        bool insertedX(true);
        bool insertedY(true);
        bool insertedZ(true);
        insertedX =
```

```
            insertedX && pointsX.setCoefficient(gid, point.x());
        insertedY =
            insertedY && pointsY.setCoefficient(gid, point.y());
        insertedZ =
            insertedZ && pointsZ.setCoefficient(gid, point.z());
}
 //Cycle for repeated nodes
    for (ID i=0; i < this->M_mesh->numVertices(); ++i)
    {
// Saving the initial mesh if M_multimesh is false
        typename MeshType::point_Type point;
        if ( this->M_multimesh )
            point = this->M_mesh->point(i);
        else
            point = this->M_mesh->pointInitial(i);
        gid = point.id();
        bool insertedX(true);
        bool insertedY(true);
        bool insertedZ(true);
        for (UInt k=0;
             k < this->M_velocityDof.M_doubledDofOnVerticesList.size();
             k++)
        {
            if (gid ==
                this->M_velocityDof.M_doubledDofOnVerticesList[k].first)
            {
                //If it's a doubled dof
                UInt gid2 = this->
                    M_velocityDof.M_doubledDofOnVerticesList[k].second;
                insertedX = insertedX &&
                    pointsX.setRepeatedCoefficient(gid, point.x(), gid2);
                insertedY = insertedY &&
                    pointsY.setRepeatedCoefficient(gid, point.y(), gid2);
                insertedZ = insertedZ &&
                    pointsZ.setRepeatedCoefficient(gid, point.z(), gid2);
            }
        }
    }
...
}
```

Capitolo 5

Risultati numerici

5.1 Cilindro con interfaccia porosa

Cominciamo ora ad analizzare i risultati numerici ottenuti utilizzando il nostro codice.

Per prima cosa confrontiamo il caso di fluido incomprimibile all'interno di un cilindro in presenza o meno di una condizione di interfaccia. Tale interfaccia è posizionata nella sezione centrale del cilindro.

In primo luogo abbiamo provveduto a costruire una mesh che si adattasse alle nostre esigenze. La mesh è stata costruita secondo i criteri descritti nella sezione 4.3.1 ed è conforme alla parete che costituisce l'interfaccia porosa.

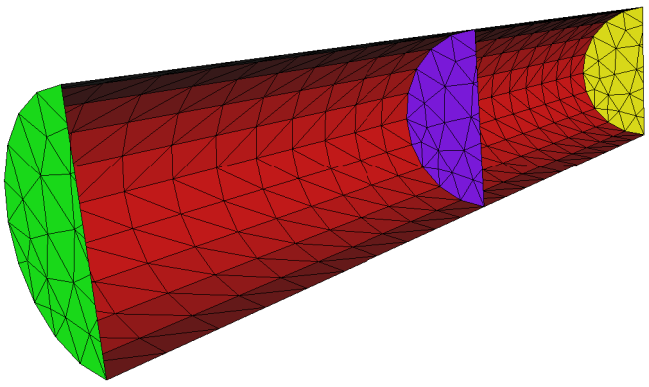

Figura 5.1: Mesh cilindica con parete di interfaccia interna; l'interfaccia corrisponde alla sezione centrale del cilindro e suddivide la mesh in due macrovolumi.

Partendo dalla mesh `tube20.mesh` già presente in LifeV ne abbiamo costruita una che avesse una sezione trasversale in cui tutte le facce avessero lo stesso marker (diverso da 0). In tale modo riusciamo a distinguere all'interno del dominio un certo numero di facce, appartenenti all'interfaccia, su cui in seguito sarà imposta la condizione di porosità. Abbiamo indicato gli elementi appartenenti all'interfaccia con un marker superiore a 1000000.

È inoltre fondamentale che la mesh che andremo ad utilizzare sia suddivisa dalla presenza dell'interfaccia in due macrovolumi, ovvero i due sottodomini generati dalla parete interna. È importante per il nostro codice che i tetraedri che compongono la mesh abbiano marker di volume differenti a seconda del sottodominio destro o sinistro a cui appartengono.

Una rappresentazione della mesh utilizzata per le nostre simulazioni è riportata in figura 5.1.

Il caso senza interfaccia corrisponde al caso già risolto nel test `test_cylinder` contenuto in LifeV.

La soluzione è quella nota: un profilo lineare per la pressione e un flusso di tipo Poiseuille per la velocità. In particolare abbiamo imposto un valore di pressione pari a 10 all'ingresso del fluido, un valore nullo di pressione all'uscita e una condizione di Dirichlet nulla sulle pareti del cilindro. Per la risoluzione numerica abbiamo utilizzato lo spazio P1Bolla–P1.

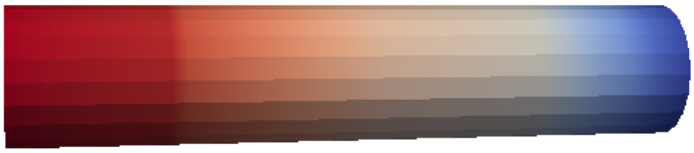

Figura 5.2: Pressione ottenuta utilizzando P1Bolla-P1 in assenza di condizioni di interfaccia.

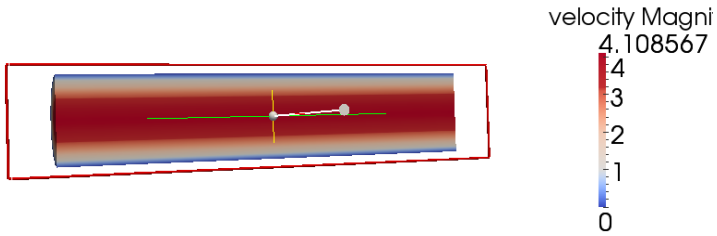

Figura 5.3: Velocità ottenuta utilizzando P1Bolla-P1 in assenza di condizioni di interfaccia.

I risultati ottenuti sono riportati nelle figure 5.2 e 5.3.

Vediamo cosa si ottiene invece in presenza di un'interfaccia porosa. Modellizziamo la presenza di questa condizione di tipo resistivo per mezzo di una condizione di Robin posta sulla parete di interfaccia.

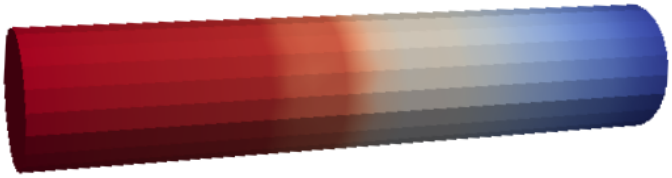

Figura 5.4: Pressione ottenuta utilizzando P1Bolla-P1 con condizione di Robin all'interfaccia.

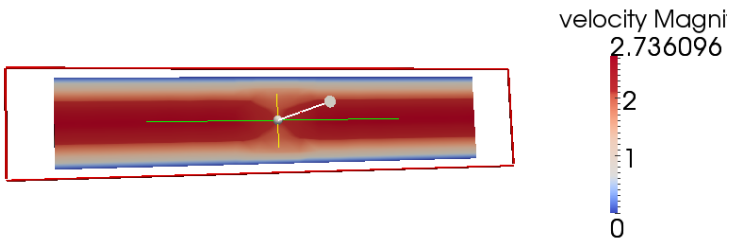

Figura 5.5: Velocità ottenuta utilizzando P1Bolla-P1 con condizione di Robin all'interfaccia.

I risultati sono quelli in figura 5.4 e 5.5. Si nota per la pressione un profilo lineare nei due tratti di tubo, mentre all'interfaccia è evidente il salto della pressione. Per quanto riguarda la velocità, prima e dopo lo stent, come ci attendiamo dalla teoria, ritroviamo un flusso di Poiseuille. In prossimità della parete, invece, il flusso da parabolico tende a diventare piatto. Ciò è evidente da quanto riportato in figura 5.6. Questi risultati confermano quelli ottenuti nell'articolo [1] che stiamo considerando. La velocità massima raggiunta dal fluido è ovviamente minore rispetto a quella che si otterrebbe in assenza dell'interfaccia porosa.

Consideriamo ora come varia il comportamento del fluido al crescere del coefficiente resistivo imposto sull'interfaccia. Come possiamo osservare in figura 5.7, maggiore è il coefficiente imposto, maggiore è il salto della pressione in corrispondenza dell'interfaccia e, di conseguenza, minore è il valore massimo della velocità che si ottiene. Tende inoltre ad allargarsi la regione in cui il flusso tende ad appiattirsi. Per un valore della resistenza pari a 0 la condizione di interfaccia diventa una condizione di Neumann e si torna al caso classico.

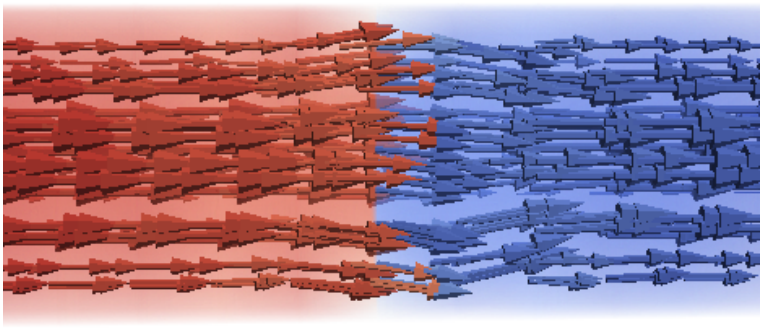

Figura 5.6: Traiettorie della velocità all'interno del cilindro con interfaccia porosa. In corrispondenza dell'interfaccia è evidente come il flusso di Poiseuille tenda ad appiattirsi, per poi ricostruirsi lontano dall'interfaccia. La scala cromatica si riferisce ai valori di pressione

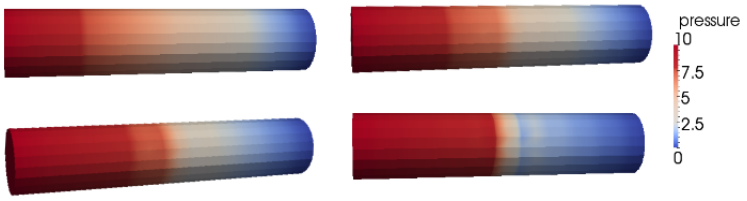

Figura 5.7: Pressione ottenuta utilizzando valori del coefficiente di resistenza pari a 1 (sinistra in alto), 2 (destra in alto), 5 (sinistra in basso) e 10 (destra in basso).

Vediamo ora cosa succede se, invece di usare lo spazio ad elementi finito discontinuo in corrispondenza della parete di interfaccia, avessimo utilizzato un comune spazio continuo per la risoluzione del problema. In tal caso la soluzione non è in grado di interpretare correttamente il salto della pressione: per avvicinarsi alla soluzione corretta sarebbe necessaria una griglia molto fitta in corrispondenza dell'interfaccia.

Mostriamo in figura 5.8 la soluzione ottenuta utilizzando spazi P1Bolla-P1 senza lo sdoppiamento dei nodi di interfaccia. È facile notare la cattiva qualità dell'approssimazione della pressione in corrispondenza dell'interfaccia. Lo spazio continuo non è in grado di cogliere opportunamente il salto e nella pressione si notano delle oscillazioni vicino all'interfaccia.

Riportiamo anche due grafici in figura 5.9 che mostrano l'andamento

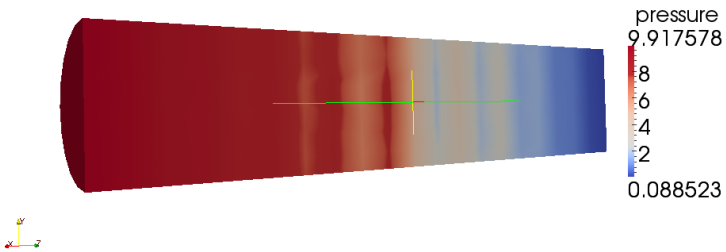

Figura 5.8: Pressione ottenuta utilizzando uno spazio P1Bolla-P1 classico con condizione di Robin all'interfaccia.

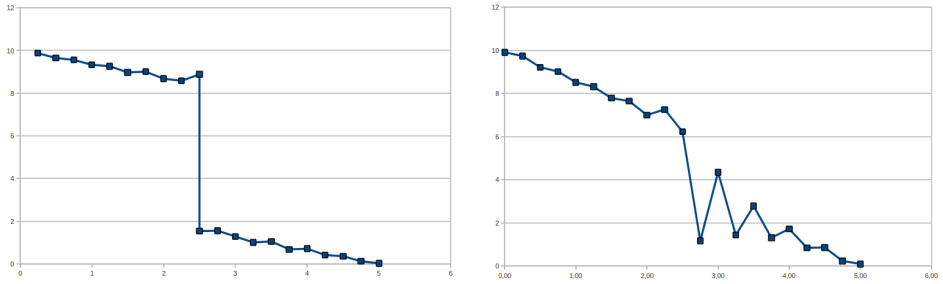

Figura 5.9: Andamento della pressione all'interno del tubo nel caso di spazi discontinui e nel caso di spazi classici.

della pressione all'interno del cilindro nei due casi analizzati: tali grafici rendono evidente la maggiore capacità dell'approssimazione con spazi localmente discontinui di rappresentare al meglio il salto della pressione.

Bisogna comunque sottolineare che anche utilizzando una griglia molto fitta per gli spazi ovunque continui il valore del flusso che si otterrebbe sarebbe più grande rispetto a quello che si otterrebbe con lo spazio discontinuo. È da notare infatti come la soluzione ottenuta con il metodo da noi implementato sia efficace anche con griglie relativamente lasche. Tutti questi risultati confermano quanto riportato nell'articolo [1] a cui ci siamo ispirati.

Concludiamo ora dicendo che il caso di cilindro con interfaccia porosa fa parte di un nuovo test da noi introdotto in LifeV, chiamato `test_cylinderWithInterface`.

5.2 Un modello di aneurisma

Proviamo ora ad analizzare i risultati numerici del nostro codice su un modello molto semplificato di aneurisma cerebrale.

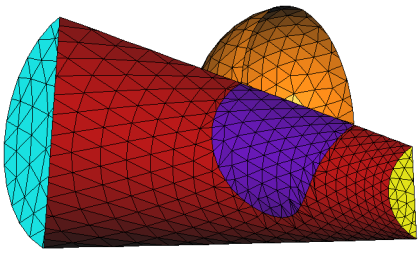

Figura 5.10: Modello semplificato di aneurisma cerebrale: mesh utilizzata per le simulazioni numeriche. La superficie in viola di separazione tra il cilindro (vaso sanguigno) e la sfera (aneurisma) rappresenta lo stent. Ringraziamo il dott. Franco Dassi per averci aiutato nella creazione di questa mesh.

La mesh che abbiamo utilizzato per le nostre simulazioni è quella rappresentata in figura 5.10. La mesh è costituita da una sfera collegata ad un cilindro: la prima rappresenta l'aneurisma, la seconda il vaso sanguigno. La superficie di separazione tra il cilindro e la sfera rappresenta la parte di stent a contatto con il flusso sanguigno. È su questa superficie che vanno applicate le condizioni di interfaccia di resistenza.

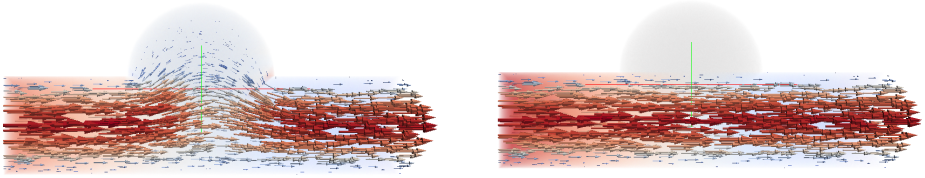

Figura 5.11: Traiettorie della velocità all'interno del modello semplificato di aneurisma in assenza (sinistra) e in presenza (destra) dello stent.

È possibile fare un confronto tra il comportamento del fluido in assenza e in presenza dello stent. In entrambi i casi, abbiamo imposto condizioni di Dirichlet nulle sulla superficie esterna del cilindro e della sfera. Abbiamo inoltre imposto un ingresso di tipo Poiseuille sulla parete di input (sinistra) e una condizione di Neumann omogenea in uscita (destra). I risultati sono riportati in figura 5.11.

È evidente come nel caso senza stent le linee di flusso tendono ad allargarsi in corrispondenza dell'aneurisma e a riempire la sfera, causando una riduzione della velocità; al termine della sfera viene poi riscostituito il flusso di Poiseuille imposto all'ingresso.

In presenza dello stent è invece evidente come la resistenza imposta all'interfaccia sia di ostacolo all'ingresso del flusso sanguigno all'interno dell'aneurisma. Per evidenziare l'effetto dovuto alla presenza dello stent abbiamo imposto un coefficiente resistivo molto alto, che impedisce quasi totalmente il passaggio del fluido.

Le basse velocità che si ritrovano all'interno dell'aneurisma a causa della presenza dello stent favoriscono la coagulazione del sangue e ne impediscono la rottura.

Nella realtà è impossibile che lo stent eserciti sul flusso sanguigno una resistenza troppo elevata. Infatti una resistenza molto alta corrisponde al caso in cui lo stent è costituito da una griglia molto fitta. Benché concettualmente ci piacerebbe avere uno stent che blocchi totalmente l'afflusso di sangue nell'aneurisma, nella pratica ci serve che esso sia abbastanza flessibile da poter essere inserito all'interno di un vaso sanguigno e potersi muovere al suo interno senza danneggiarlo fino a raggiungere la posizione desiderata. Per garantire sufficiente flessibilità occorre che la griglia che costituisce lo stent non sia troppo fitta.

Ciò suggerisce l'idea di applicare sugli elementi di interfaccia una resistenza non omogenea in modo da studiare la disposizione ottimale delle maglie dello stent. Si tratterebbe di un problema vincolato: si cerca di minimizzare la velocità del sangue all'interno dell'aneurisma pur garantendo al contempo una sufficiente flessibilità delle pareti dello stent.

Questa è una possibile applicazione futura del codice da noi sviluppato che potrebbe essere di grande interesse nel campo biomedico.

5.3 Un'analisi di convergenza

Testiamo ora i tassi di convergenza dello schema che abbiamo implementato su un caso test analitico. Usiamo spazi P1Bolla per la velocità e P1 per la pressione; questa è una coppia che dalla teoria classica sappiamo essere stabile. Vogliamo testare se l'utilizzo di spazi con discontinuità all'interfaccia conserva le proprietà di convergenza di tali spazi.

Cerchiamo di costruire una soluzione analitica su cui valutare i tassi di convergenza. Ci ispiriamo al test proposto in [1] nel caso bidimensionale; estendiamo tale analisi al caso 3D imponendo un andamento della velocità costante (in particolare nullo) nella terza componente.

Sia $\Omega = (0,2) \times (0,1) \times (0,0.2)$ il dominio del fluido, suddiviso in due sottodomini $\Omega_1 = (0,L) \times (0,1) \times (0,0.2)$ e $\Omega_2 = (L,2) \times (0,1) \times (0,0.2)$ da un'interfaccia $\Gamma = \{L\} \times (0,1) \times (0,0.2)$ con $L = 1$. Assumiamo un

valore per la viscosità $\mu = 0.04$ e definiamo la resistività per mezzo della matrice $\mathbf{R}_\Gamma = 100\mathbf{I}$. Fatte queste ipotesi, possiamo calcolare il valore di $\mathbf{f}$ e le opportune condizioni al contorno affinché le seguenti funzioni siano soluzione del problema di Stokes (3.13):

$$\mathbf{u}_1 = \begin{bmatrix} -19.98x + 10x^2 \\ -40.04 + 19.98y + 40x^2 - 20xy \\ 0 \end{bmatrix}, \mathbf{u}_2 = \begin{bmatrix} 25 - 69.98x + 35x^2 \\ 9.96 + 69.98y - 10x^2 - 70xy \\ 0 \end{bmatrix}$$

$$p_1 = 800y^2, \qquad p_2 = 998 + 800y^2$$

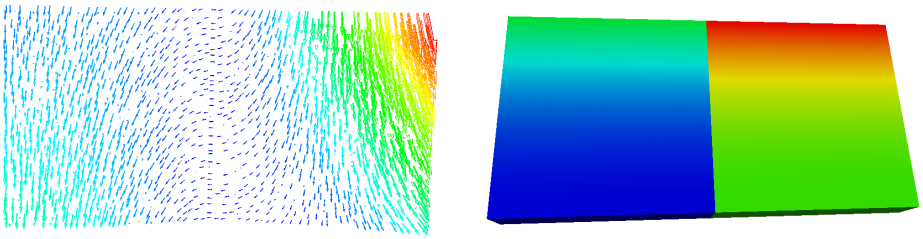

Figura 5.12: Soluzione esatta in velocità e pressione per il caso test in esame. I valori vanno da 9.63 a 103 per la velocità e da 0 a 1800 per la pressione.

Tale soluzione analitica è stata scelta come polinomiale di grado 2 in x e y e in modo tale da garantire un grosso salto di pressione all'interfaccia e un salto moderato delle derivate normali della velocità. Una rappresentazione della soluzione esatta in velocità e pressione è riportata in figura 5.12.

Per le simulazioni abbiamo utilizzato diverse mesh regolari con numero crescente di elementi (circa il doppio da una mesh all'altra). Poiché possiamo considerare il volumetto del singolo elemento della mesh proporzionale ad h^3, si ha che il parametro h decresce da una mesh all'altra in maniera proporzionale a $\sqrt[3]{2}$

I risultati sono riportati in tabella:

Num.Elementi	$\|\mathbf{u} - \mathbf{u}_h\|_{L^2}$	$\|p - p_h\|_{L^2}$
1837	0.0775757	0.526151
3753	0.0429816	0.351991
7401	0.0282139	0.229406
14386	0.0171432	0.148667

Come si può notare dal grafico riportato in figura 5.13, il tasso di convergenza che si ottiene utilizzando spazi con discontinuità localizzata sull'interfaccia è quadratico per l'errore in norma L^2 sia per quanto riguarda la velocità sia per quanto riguarda la pressione. Questo risultato conferma quanto ottenuto in [1] per l'analogo problema nel caso bidimensionale.

Possiamo sottolineare come i nostri spazi localmente discontinui con gradi di libertà raddoppiati all'interfaccia non peggiorino in alcun modo

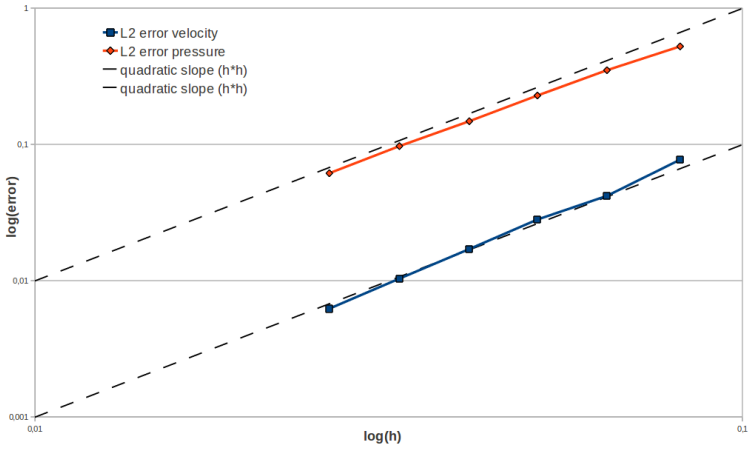

Figura 5.13: Grafici degli errori in scala logaritmica.

le proprietà di convergenza della coppia di spazi compatibili che stiamo considerando (nel nostro caso P1Bolla–P1).

C'è da notare che ci troviamo in un caso di superconvergenza per quanto riguarda la pressione, per la quale ci aspetteremmo un tasso lineare con h. Per provare a dare una spiegazione a questo risultato inatteso consideriamo i risultati di convergenza per il problema di Stokes stazionario. Introduciamo lo spazio delle funzioni a divergenza discreta nulla:

$$V_{h,div} = \{v_h \in V_h : \ b(v_h, q_h) = 0, \forall q_h \in Q_h\}$$

Vale allora il seguente risultato (vedere [8]):

$$\|p - p_h\|_{L^2} \leq C(\inf_{v_h \in V_{h,div}} \|u - v_h\|_{H^1} + \inf_{q_h \in Q_h} \|p - q_h\|_{L^2})$$

Se però gli spazi V_h e Q_h soddisfano la condizione inf–sup discreta

$$\inf_{q \in Q_h, q_h \neq 0} \sup_{v_h \in V_h} \frac{b(v_h, q_h)}{\|v_h\|_{H^1} \|q_h\|_{L^2}} \geq \beta_h > 0 \qquad \forall h$$

si dimostra la seguente stima per la pressione:

$$\|p - p_h\|_{L^2} \leq \frac{2\nu}{\beta_h} \inf_{v_h \in V_{h,div}} \|u - v_h\|_{H^1} + (1 + 2\frac{\sqrt{d}}{\beta_h} \inf_{q_h \in Q_h} \|p - q_h\|_{L^2} \quad (5.1)$$

dove ν è la viscosità cinematica e d è la dimensione del problema (nel nostro caso 3). Nel caso β_h dipenda da h, ciò comporterebbe la non ottimalità dell'approssimazione di Galerkin e in casi drammatici una non convergenza. Per i particolari spazi ad elementi finiti che stiamo considerando non esiste

ancora una dimostrazione della validità dell'inf–sup discreta e dell'indipendenza di β_h da h. Tuttavia i risultati numerici da noi ottenuti nel problema in esame sembrano confermare la validità di tali ipotesi.

Ricordiamo che nel caso di spazi ad elementi finiti P1Bolla–P1 il primo termine di errore è dell'ordine di h, mentre il secondo è dell'ordine di h^2.

Nel nostro problema tuttavia i valori relativi alla pressione sono molto più grandi rispetto a quelli relativi alla velocità e di conseguenza anche le differenze che influenzano la stima di convergenza del termine in pressione hanno scale diverse. Per questo motivo nella nostra analisi di convergenza non riusciamo a cogliere il termine legato alla velocità ma soltanto quello legato alla pressione riscontrando quindi un andamento quadratico.

Questa è soltanto una possibile spiegazione di questa anomalia ma non è certo il nostro primo interesse. Quello che ci preme è sottolineare come l'introduzione di una fissione nella mesh in corrispondenza dell'interfaccia non peggiori le proprietà di convergenza ottimali della coppia di elementi finiti che stiamo considerando.

Capitolo 6

Conclusioni

Nel nostro lavoro abbiamo introdotto una modellizzazione dello stent come interfaccia porosa. Dal punto di vista analitico questo consiste nell'imporre una condizione di Robin di tipo resistivo su un'interfaccia interna al dominio di calcolo. Tale approccio permette di evitare l'uso di griglie troppo fitte che ricostruiscano la geometria dello stent.

Per permettere di leggere correttamente il salto della pressione in corrispondenza dell'interfaccia interna viene proposto inoltre l'uso di spazi ad elementi finiti che siano continui ovunque e discontinui in corrispondenza dell'interfaccia. Ciò significa che i gradi di libertà in corrispondenza dell'interfaccia interna vengono duplicati.

Il nostro lavoro è consistito nell'introdurre all'interno di LifeV tutte le modifiche necessarie per la risoluzione di problemi con interfaccia interna.

Abbiamo poi utilizzato il codice che abbiamo generato per risolvere un caso di test di flusso di Navier–Stokes incomprimibile all'interno di un cilindro con una interfaccia porosa all'interno del dominio.

Abbiamo poi verificato il comportamento della soluzione per diversi valori del coefficiente di resistenza. Abbiamo inoltre confrontato la soluzione ottenuta con il caso classico di cilindro senza interfaccia.

Abbiamo inoltre osservato la differenza tra la soluzione ottenuta con spazi localmente discontinui e quella che si ottiene utilizzando spazi ovunque continui: i risultati ottenuti nel secondo caso risultano meno accurati anche nei casi in cui abbiamo utilizzato griglie molto più fitte.

Abbiamo anche proposto una simulazione numerica su un modello semplificato di aneurisma cerebrale, per mostrare l'effetto dello stent sul flusso sanguigno.

Infine abbiamo verificato che i nostri spazi ad elementi finiti discontinui in corrispondenza dell'interfaccia mantengono le proprietà di convergenza degli spazi classici da cui hanno origine. Nel nostro caso abbiamo utilizzato spazi P1Bolla per la velocità e P1 per la pressione.

In conclusione, la tecnica proposta nell'articolo di modellizzare lo stent come condizione interna di interfaccia porosa e utilizzare spazi ad elementi finiti con discontinuità locale sembra efficace; essa permette di ottenere risultati accurati pur con un costo computazionale molto ridotto rispetto alle tecniche classiche che richiedono una mesh molto fitta che ricostruisca fedelmente la geometria dello stent. L'utilizzo di tali spazi permette inoltre di cogliere con efficacia il salto della pressione in corrispondenza dell'interfaccia.

Il codice è stato sviluppato a partire da un modello fisico ben definito, quello dello stent, ma a nostro avviso lo stesso può essere utilizzato per risolvere un'ampia classe di problemi di fluidodinamica, e non solo, che richiedono la presenza di interfacce interne.

6.1 Sviluppi futuri

Per portare a termine il nostro lavoro abbiamo dovuto intervenire profondamente all'interno dei codici di LifeV su vari livelli. Abbiamo dovuto risolvere diversi tipi di problematiche relative ai codici di LifeV che in precedenza non erano emerse. Giunti al termine del nostro lavoro, ci sembra interessante fare in modo che il nostro codice possa diventare una nuova branch di LifeV in modo da renderne disponibile l'utilizzo a chiunque ne avesse bisogno.

Per fare ciò occorre però risolvere alcuni problemi. Parte di questi riguardano la compatibilità con altri esempi e test già presenti nella versione di base di LifeV. Le modifiche più importanti da compiere riguardano però il fatto che il nostro codice non funziona correttamente nel caso di calcolo parallelo. Il problema principale sembra legato all'assegnazione dei due gradi di libertà nei nodi dell'interfaccia. In quest'ottica occorre dunque apportare alcune migliorie al codice per fare in modo che il nostro lavoro possa essere inserito nella versione parallela di LifeV.

Inoltre sarebbe interessante estendere queste modifiche ad altri tipi di solutore o ad altri tipi di condizione interna. Il nostro codice infatti funziona nel caso di domini separati da una parete di interfaccia in due sottodomini disgiunti. Si potrebbe anche considerare il caso in cui siano presenti più pareti di interfaccia o in cui l'interfaccia non separi completamente il dominio.

Non dobbiamo infine dimenticare come l'idea alla base del nostro lavoro sia partita dalla simulazione numerica del flusso sanguigno all'interno di un aneurisma cerebrale e in presenza dello stent. Sarebbe molto interessante applicare il nostro codice su una geometria realistica e confrontare i risultati numerici ottenuti con ciò che si otterrebbe dall'osservazione diretta della realtà.

Una ulteriore applicazione che potremmo pensare per il nostro codice è quella di effettuare una ottimizzazione a basso costo dello stent. Questa idea può essere sviluppata imponendo una resistenza sull'interfaccia non

omogenea, ma variabile su ciascun elemento che compone la parete dello stent. È possibile a questo punto risolvere un problema vincolato e calcolare la distribuzione ottimale della resistenza e, dunque, la disposizione ottimale delle maglie dello stent in modo da ottimizzare il flusso sanguigno all'interno dell'aneurisma. Un possibile vincolo è legato alle caratteristiche fisiche dello stent; esso infatti deve essere abbastanza flessibile da potersi muovere all'interno di vasi sanguigni molto piccoli fino a raggiungere il punto in cui deve essere posizionato. Una concentrazione troppo elevata di maglie potrebbe rendere impossibile tale operazione.

In quest'ottica si potrebbe poi pensare l'implementazione di un'interfaccia con resistenza non omogenea per studiare l'efficacia di uno stent in base alla disposizione delle maglia che lo compongono. Si potrebbe quindi implementare un procedimento di ottimizzazione di forma degli stent basato sulla distribuzione della resistenza nell'interfaccia porosa.

Bibliografia

[1] M.A. Fernandez, J.-F. Gerbeau, V. Martin, *Numerical Simulation of Blood Flows Through a Porous Interface*. ESAIM: M2AN 42, pp. 961–990, 2008

[2] A. Caiazzo, M.A. Fernandez, J.-F. Gerbeau, V. Martin, *Projection Schemes for Fluid Flows Through a Porous Interface*. SIAM J. Sci. Comput., Vol. 33, Num. 2, pp. 541–564, 2011

[3] L. Augsburger, P. Reymond, D.A. Rufenacht, N. Stergiopulos, *Intercranial Stents Being Modeled as a Porous Medium: Flow Simulation in Stented Cerebral Aneurysms*. Annals of Biomedical Engineering, Vol. 39, Num. 2, pp. 850–863, 2011

[4] R. Popescu, *LifeV - Developer Manual*. disponibile su internet (http://www.lifev.org), 2011

[5] R. Popescu, *LifeV - User Manual*. disponibile su internet (http://www.lifev.org), 2011

[6] L. Formaggia, *LifeV - Development Guidelines*. disponibile su internet (http://www.lifev.org), 2011

[7] H. Si, *TetGen, A Quality Tetrahedral Mesh Generator and Three-Dimensional Delauney Triangulator - Version 1.4 - User's Manual*. disponibile su internet (http://tetgen.berlios.de), 2006

[8] A. Quarteroni, *Modellistica Numerica per Problemi Differenziali*, Springer–Verlag, Milano, 2006

[9] L. Formaggia, F. Saleri, A. Veneziani, *Applicazione ed Esercizi di Modellistica Numerica per Problemi Differenziali*, Springer–Verlag, Milano, 2005

[10] F. Brezzi, M. Fortin, *Mixed and Hybrid Finite Elements Methods*, Springer–Verlag, New York, 1991

[11] M. Astorino, *Interaction Fluide–Structure dans le Système Cardio-vasculaire. Analyse Numérique et Simulation*, Tesi di Dottorato in Matematica Applicata, Université Pierre et Marie Curie - Paris VI, 2010

[12] A.J. Chorin, J. Marsden, *A Mathematical Introduction to Fluid Mechanics*, Springer–Verlag, 3 ed., 1990

[13] H.C. Elmann, D.J. Silvester, A.J. Wathen, *Finite Elements and Fast Iterative Solvers with Applications in Incompressible Fluid Dynamics*, Oxford University Press, 2005

[14] A. Ern, J.L. Guermond, *Theory and Practice of Finite Elements*, Springer–Verlag, 2004

[15] V. Girault, P.A. Raviart, *Finite Elements Methods for Navier–Stokes Equations. Theory and Algorithms*, Springer–Verlag, 1986

[16] A. Quarteroni, A. Valli, *Numerical Approximation of Partial Differential Equations*, Springer–Verlag, 1997

Ringraziamenti

Giunti a questo punto sono tante le persone che vogliamo ringraziare: primi tra tutti i nostri genitori, Matteo e Chiara che ci hanno aiutato e sostenuto lungo questo difficile percorso fino ad oggi. Se non fosse stato per il loro incondizionato appoggio difficilmente saremmo giunti a questo importante obiettivo.

Un sincero ringraziamento lo vogliamo rivolgere al Prof. Nobile e al Prof. Formaggia che ci hanno aiutato nell'elaborazione e nella stesura di questo elaborato e insieme a loro anche al Prof. Dubini che si è prestato nella parte di organizzazione del progetto e che aveva dato la sua disponibilità ad accompagnarci in laboratorio per una verifica sperimentale dei nostri risultati, che purtroppo non siamo riusciti ad eseguire. Vogliamo ringraziarli per l'attenzione che ci hanno dedicato in questi mesi, per la disponibilità e per la cortesia con cui ci hanno sempre accolto nei loro uffici anche negli orari più scomodi.

Un grazie enorme lo vogliamo dire ai dottorandi del "Tender", in particolare a Franco Dassi per le sue fantastiche mesh (vedi 5.10) che ci hanno permesso di ottenere i bei risultati proposti, ad Antonio Cervone e Alessio Fumagalli che ci hanno aiutato nel difficile apprendimento e utilizzo di LifeV e a tutti gli altri, che ci hanno accolto e ospitato nel loro splendido gruppo di lavoro.

A proposito di gruppi di lavoro non possiamo non ringraziare tutto il gruppo LifeV-dev che si occupa dello sviluppo del codice di calcolo che abbiamo utilizzato e che ci ha reso partecipi di tutte le riunioni in cui si è discusso dello sviluppo del software.

Infine una moltitudine di grazie la vogliamo dire a tutti i nostri compagni e amici che ci hanno fatto compagnia durante questi anni. Vogliamo ringraziarli per ogni singolo pomeriggio speso insieme a studiare, per ogni ora di corso che hanno seguito con noi, per ogni esame che hanno affrontato al nostro fianco e per ogni partita di briscola o scopone in cui ci siamo affrontati. E' sicuramente merito loro se in questi anni le grigie giornate milanesi sono state più divertenti e piacevoli.