Laurea Magistrale
in Ingegneria Matematica

Progetto per il corso di
Programmazione Avanzata per il Calcolo Scientifico
Professor Luca Formaggia


# Implementazione di un Solutore per Flussi in Mezzi Porosi con Fratture

Progetto di:
Francesco Della Porta
Matr. 783358

Anno Accademico 2012–2013

# Indice

# Capitolo 1

# Introduzione

Il problema di Darcy che modellizza flussi in mezzi porosi è un soggetto di ricerca molto importante ed è applicato in molti ambiti. L'applicazione piú diffusa è probabilmente quella che studia flussi sotto il suolo terrestre, primo fra tutti quelli di petrolio. Il problema, già di per sé complicato, in questo tipo di applicazioni è spesso reso più difficile dalla presenza di fratture nel sottosuolo; nelle fratture il fluido si muove con proprietà molto diverse rispetto alle zone porose. I flussi di questo tipo vengono chiamati flussi in mezzi porosi con fratture. Esistono varie metodologie in letteratura per trattare numericamente questi problemi; nel lavoro qui presentato si è implementato il metodo sviluppato da Karimi-Fard, Durlofsky e Aziz in [1] in un ambito di discretizzazione a volumi finiti.
In questo progetto si è partiti da una mesh esistente, in cui le fratture sono già state discretizzate, e sono stati implementati i seguenti tool per la risoluzione di un flusso di Darcy in mezzi porosi con fratture:

- Innanzitutto è stata implementata una nuova classe per contenere la mesh. Quella esistente, infatti, permetteva di apportare cambiamenti alla griglia con grande facilità, ma, d'altro canto, non consentiva di avere accesso ai dati con la desiderata facilita'. La struttura adottata permette una buona compatibilità ed efficienza anche con problemi diversi dal problema di Darcy in mezzi porosi con fratture e anche con metodi di discretizzazioni diversi, quali le differenze finite mimetiche.

- È stata implementata una classe per contenere i bordi di un dominio poligonale ed una classe per tenere le condizioni al contorno, in modo tale da poter imporre sia condizioni di tipo Neumann, che condizioni di tipo Dirichet, omogenee e non.

- È stata implementata una classe che, partendo dal lavoro descritto nei punti precedenti, permette di costruire la matrice di stifness per il problema di Darcy con il metodo riportato in [1].

- È stata implementata una classe che permette di costruire la matrice di massa per la griglia data.
- È stata implementata una classe che permette di calcolare la norma in $L^2$ della soluzione e di discretizzare una forzante data in forma continua.

È importante notare sin da ora, sebbene una versione 3D della griglia da cui siamo partiti non sia stata ancora implementata, tutti i tool imlementati in questo progetto sono compatibili sia con il caso bidimensionale che con il caso tridimensionale.

I capitoli che seguiranno tratteranno i seguenti argomenti:

- Il prossimo spiega brevemente la natura fisica del problema e il metodo di discretizzazione riportato in [1].
- Il capitolo successivo illustra le classi implementate e le relative scelte implementative.
- Nell'ultimo capitolo vengono riportati alcuni risultati di test numerici eseguiti con il codice qui descritto.

# Capitolo 2

# Flussi di Darcy in mezzi porosi con fratture

Questo capitolo è diviso in due sezioni: nella prima si introducono i flussi di Darcy in mezzi porosi con fratture; nella seconda viene poi presentato il metodo di discretizzazione implementato.

## 2.1 Il modello matematico

Il problema di descrivere il moto di un fluido in un mezzo poroso, ossia lo spostamento di un liquido o di un gas all'interno di un solido con piccoli interstizi che ne consentono il moto, sorge in molte branche della scienza applicata e dell'ingegneria. Fra i vari campi di applicazione, quali la biomatematica, quella più importante riguarda la descrizione del moto di fluidi sotto terra. Quest'ultima è materia di studio per l'idrologia, come per le multinazionali del petrolio che da anni finanziano questo ramo della ricerca, al fine di ottimizzare il processo di estrazione del greggio.

Generalmente, lo studio matematico dei fluidi, del loro campo di moto e del relativo campo di pressione, viene condotto tramite la risoluzione numerica delle equazioni di Navier-Stokes, nella loro versione comprimibile e non. Tuttavia, quando si considerano dei flussi in mezzi porosi, dal punto di vista numerico si crea la necessità di risolvere non solo la scala grande, che è quella del mezzo poroso nella sua totalità, ma anche quella cosiddetta piccola che è quella degli interstizi nel corpo solido. Spesso queste due scale sono diverse fra loro di parecchi ordini di grandezza e questo comporterebbe la necessità di una griglia computazionale troppo fitta e, conseguentemente, tempi computazionali troppo elevati per essere presi in considerazione. Si pensi a tal proposito ad un flusso d'acqua in una montagnetta di sabbia: la scala dello spazio fra un granello e l'altro è di molto inferiore alla scala della

montagnetta stessa.

Henry Darcy aveva in passato studiato questo problema fisico dal punto di vista sperimentale, arrivando a formulare una legge simile a quella di Fourier per la conduzione del calore e a quella di Fick per i processi di diffusione. Il risultato da lui ottenuto è il seguente:

$$\rho \mathbf{u} = -\frac{k}{\mu}(\nabla p - \rho g) \tag{2.1}$$

dove $k$ è la permeabilitá del mezzo poroso, $p$ e $\mathbf{u}$ sono rispettivamente la pressione e la velocità del fluido, $\rho$ e $\mu$ rispettivamente la sua densità e la sua viscosità e $g$ è la costante gravitazionale.

Nonostante questa equazione sia stata ricavata sperimentalmente, esistono diversi procedimenti che cercano di legittamarla in maniera più o meno formale . Per esempio, è possibile ricavare l'equazione di Darcy tramite un metodo di omogeneizzazione partendo dall'equazione di Stokes, oppure si può arrivare allo stesso risultato utlizzando la teoria delle miscele. Entrambe questi procedimenti modellizzano il fenomeno ad una sua mesoscala, ossia una scala intermedia fra quella grande del dominio, e quella piccola delle porosità, ed è a questa scala che il fenomeno è descritto dalle equazioni di Darcy. Un punto comune a tutte le teorie sono le condizioni fisiche di validità del modello: l'equazione di Darcy infatti è applicabile solo nel caso in cui la grande scala e la piccola scala differiscano di vari ordini di grandezza. Inoltre è importante che la velocità del fluido non sia eccessivamente elevata rispetto alla porosità del terreno; tuttavia, fortunatamente, questa condizione è quasi sempre verificata.

Il problema che si incontra nella maggior parte delle applicazioni è quello di caratterizzare la porosità del mezzo. Questo dato può essere molto complicato da ricavare sperimentalmente, soprattutto perchè spesso il mezzo è anisotropo e non omogeneo. In questo caso infatti $k$ diventa un campo tensoriale. Noi, nel seguito, prenderemo in considerazione semplicemnete il caso in cui la permeabilità $k$ è non omogenea ma isotropa e quindi $k = k(x)$. In molte applicazioni, si è soliti considerare il fluido come se fosse incomprimibile o semi-comprimibile. Per l'acqua, ad esempio, questa ipotesi è abbastanza standard. In questo lavoro ci si concentra sul caso in cui il fluido è incomprimibile, lasciando a possibili sviluppi futuri il caso semi-comprimibile. Dalla teoria della meccanica dei continui sappiamo che questa ipotesi ci obbliga a includere nel problema il seguente vincolo di incomprimibilità:

$$div(\mathbf{u}) = 0. \tag{2.2}$$

D'altro canto, grazie all'incomprimibilità del fluido, e ipotizzando di considerare un fluido monofase, possiamo trattare la densità del fluido $\rho$ come

una costante. Riscalando opportunamente il campo di pressione otteniamo così le equazioni di Darcy nel caso incomprimibile:

$$\begin{cases} \mathbf{u} = -K\nabla p \\ div(\mathbf{u}) = 0 \end{cases} \tag{2.3}$$

dove $K = K(x)$ e dipende dalla mobilità e dalla densità del fluido e dalla permeabilità del mezzo. Per risolvere questo sistema di equazioni spesso si procede come segue: si prende la divergenza della prima equazione e si sfrutta il vincolo di incomprimibilità ottenendo

$$-div(K\nabla p) = 0, \tag{2.4}$$

che è un'equazione ellittica facilmente risolvibile numericamente a patto di supporre $K(x) \geq c$ per ogni $x$ con $c > 0$. Si noti che sotto la precedente ipotesi, fisicamente sempre verificata, e nel caso in cui esista almeno un piccolo tratto di bordo in cui le condizioni al contorno sono di tipo Dirichlet, si dimostra grazie al teorema di Lax-Milgram (si veda [4] per esempio) che il problema (2.4) ammette una ed una sola soluzione e il problema è ben posto.

Nelle applicazioni idrologiche e, più in generale, nello studio del moto dei fluidi nel sottosuolo, spesso il problema è complicato dalla presenza di fratture nel mezzo poroso in cui la permeabilità è molto più alta e quindi il fluido si sposta con velocità molto maggiore. Queste zone appaiono, solitamente, come molto lunghe e molto sottili rispetto alla dimensione del dominio, ma infinitamente più grandi rispetto ai piccoli interstizi che rendono il materiale poroso. In queste zone non c'è materiale solido ma solo fluido. Un esempio di frattura in mezzo poroso può essere una falda acquifera.

Generalmente, le equazioni di Darcy non possono essere risolte analiticamente, a meno di trattare casi il cui interesse applicativo è pressoché nullo. Per poter ottenere una soluzione si procede quindi per via numerica discretizzando il problema e risolvendolo con un calcolatore.

## 2.2 Approssimazione numerica dell'equazione di Darcy

Visto il vasto spettro di applicazione delle equazioni di Darcy, sono stati sviluppati svariati metodi per risolvere numericamente questo problema: sia discretizzando la sua forma originale (2.3), che discretizzando la sua versione ellittica (2.4). Fra le tecniche per discretizzare questo problema si possono annoverare gli elementi finiti, i volumi finiti, le differenze finite e le differenze finite mimetiche.

Gli elementi finiti, purtroppo, non garantiscono la conservazione della massa nel caso in cui il flusso sia bifase e la permeabilità sia fortemente eterogenea. Gli elementi finiti discontinui garantiscono questa proprietà ma spesso sono computazionalmente costosi. Si è quindi scelto di implementare uno schema a volumi finiti che, nonostante sia meno preciso degli altri due metodi, garantisce conservazione della massa e bassi costi computazionali.

Seguendo quanto spiegato in [3], accenniamo brevemente in cosa consiste lo schema numerico dei volumi finiti. Data un'equazione di diffusione e reazione non omogenea del tipo

$$-div(K(x)\nabla p) + \alpha p = f, \qquad \forall x \in \Omega \subset \mathbb{R}^3, \tag{2.5}$$

con $\alpha \in \mathbb{R}^+$, il primo passo da fare è suddividere il dominio $\Omega$ in un insieme finito di poliedri $\Omega_i \subset \Omega$, $i = 1 : N$ di diametro inferiore ad $h$ tali che $\cup_i \bar{\Omega}_i = \bar{\Omega}$ e che $\Omega_i \cap \Omega_j = \oslash$ per $i \neq j$; questi poliedri vengono detti volumi o celle. A questo punto si integra l'equazione (2.5) su ogni volumetto $\Omega_i$ per ottenere il seguente sistema di equazioni:

$$-\int_{\Omega_i} div(K\nabla p) + \int_{\Omega_i} \alpha p = \int_{\Omega_i} f, \qquad i = 1 : N. \tag{2.6}$$

Utilizzando il teorema della divergenza il primo termine diventa

$$-\int_{\Omega_i} div(K\nabla p) = -\int_{\partial\Omega_i} K\nabla p = -\sum_{j=1}^{L_1} \int_{l_{ij}} K\nabla p \cdot n_{ij}, \qquad i = 1 : N,$$

dove $L_i$ rappresenta il numero di lati del poligono $\Omega_i$, $l_{ij}$ il lato che separa il poligono $i$ dal poligono $j$ e $n_{ij}$ è la normale uscente dal poligono $\Omega_i$ sul lato $l_{ij}$. In questo lavoro verrà utilizzato uno schema di tipo cell-centered, ossia si approssima la soluzione $p$, come se fosse costante su ogni volume $\Omega_i$ ed il suo valore sia il valore di $p$ al centro della cella. Il problema può quindi essere facilmente essere riscritto come

$$-\sum_{j=1}^{L_1} \int_{l_{ij}} K\nabla p \cdot n_{ij} + m_i p_i = f_i, \qquad i = 1 : N$$

dove $f_i$ è un'opportuna approssimazione dell'integrale di $f$ su $\Omega_i$ e dove $m_i$ è il volume della cella. A questo punto manca la discretizzazione del termine di bordo; gli schemi a volumi finiti possono differire fra di loro per il modo in cui viene discretizzato il termine

$$\int_{l_{ij}} K\nabla p \cdot n_{ij},$$

la cui approssimazione viene spesso denotata con $H_{ij}$, chiamato comunemente flusso numerico. Lo schema di discretizzazione implementato in questo

lavoro è quello che descritto da Karimi-Fard, Durlofsky e Aziz in [1], ed è uno schema a due punti, ossia in cui

$$H_{ij} = H_{ij}(u_i, u_j).$$

Discretizziamo quindi il flusso numerico come

$$H_{ij} = T_{ij}(p_i - p_j) \tag{2.7}$$

dove $T_{ij}$ è la parte geometrica della trasmissibilità, calcolata come

$$T_{ij} = \frac{\beta_i \beta_j}{\beta_i + \beta_j} \tag{2.8}$$

e dove

$$\beta_i = \frac{A_i K_i}{D_i} \mathbf{n_i} \cdot \mathbf{f}_i. \tag{2.9}$$

In quest'ultima formula $A_i = A_j$ rappresenta l'area del lato $l_{ij}$ di interfaccia fra $\Omega_i$ e $\Omega_j$, $K_i$ è un' approssimazione di $K$ nel volume $\Omega_i$ e $D_i$ rappresenta la distanza fra il centro della cella $\Omega_i$ e il centro dell'interfaccia $l_{ij}$. Inoltre $\mathbf{n}_i = -\mathbf{n}_j$ rappresenta la normale a $l_{ij}$ uscente da $\Omega_i$ e $\mathbf{f}_i$ è il versore nella direzione che dal centro di $l_{ij}$ va al centro di $\Omega_i$. Per maggiore chiarezza si veda la figura (2.1) tratta da [1], che rappresenta le grandezze in gioco nel caso bidimensionale.

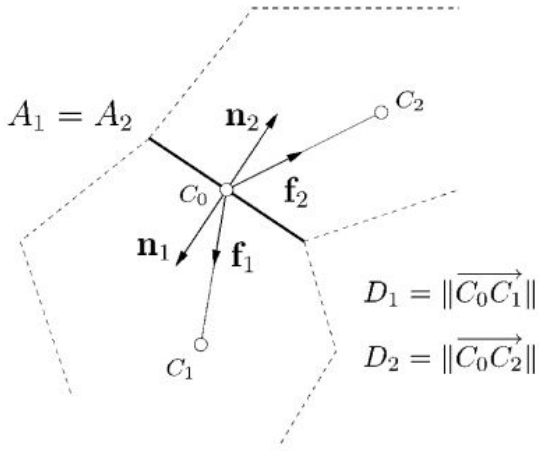



**Figura 2.1:** Grandezze che compaiono nella parte geometrica della trasmissibilità $T_{ij}$ nel caso 2D

Anche per il caso in cui il dominio è fratturato sono state sviluppate numerose tecniche di discretizzazione. La tecnica più semplice consiste nel trattare

le fratture come zone del dominio in cui la permeabilità è molto più alta. Tuttavia, nel caso in cui le fratture sono poche ma dominano il flusso, questo approcio perde di efficacia. Per questo motivo sono molto diffusi i metodi in cui le fratture vengono rappresentate individualmente. Con questo approccio la possibilità di avere una griglia strutturata risulta quasi impossibile, a meno di restringersi a casi di scarso valore applicativo. Tuttavia anche un generatore di griglie non strutturate può non riuscire nell'intento se le fratture sono tante e ravvicinate. L'approccio utilizzato, e adottato anche in [1], è quello di non raffigurarne lo spessore nel dominio ma rappresentarle come l'insieme di bordi dei volumi di controllo $\Omega_i$. Nel caso bidimensionale avremo quindi fratture che sono rappresentate come segmenti, nel caso tridimensionale da piani. Si veda a tal proposito la figura (2.2). Per calcolare

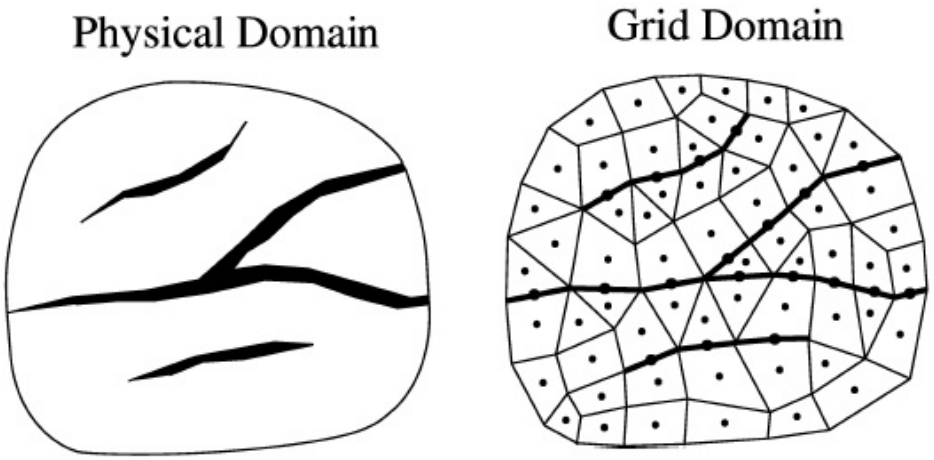



**Figura 2.2:** Esempio di approssimazione di fratture nel caso 2D

$T_ij$ fra una frattura ed una cella di volume, basta considerare la frattura come se fosse una cella e quindi con il proprio spessore; a questo punto è possibile utilizzare le formule (2.8) e (2.9) tenendo conto che il prodotto scalare $\mathbf{n_i} \cdot \mathbf{f_i} = 1$ se $i$ è l'indice relativo alla frattura. Per quanto riguarda il flusso numerico fra due celle di frattura, l'approcio usato comunemente è quello di introdurre una cella fittizia che ha lo scopo di intermediare gli scambi di flusso fra le fratture. Tuttavia queste celle fittizie sono molto piccole rispetto alle altre, rendendo pertanto il problema mal condizionato e con tempi computazionali elevati. Per ovviare a questo problema, in [1] viene introdotta un'approssimazione che ci porta a calcolare il flusso fra una frattura e l'altra $T_{ij}$ ancora con le formule (2.8) e (2.9). Tuttavia qui $D_i$ è uguale all'apertura della frattura e, nel caso bidimensionale, $\mathbf{n_i} \cdot \mathbf{f_i} = 1$ per entrambe le fratture.
Può accadere però, che più di due fratture si intersechino nello stesso punto in 2D, o nello stesso segmento in 3D. In tal caso, purtroppo, la formula usa-

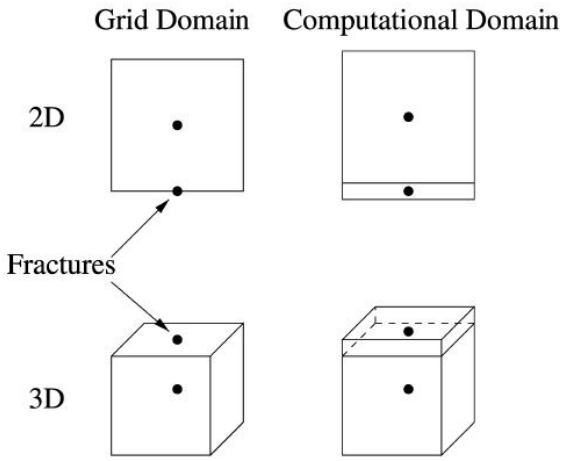



**Figura 2.3:** Esempio di trattamento di celle di fratture nel caso 2D e 3D, tratto da [1]

ta precedentemente perde significato. Per risolvere questo problema, in [1], si ricorre alla teoria dell'elettrotecnica e viene usata l'equivalenza stella-triangolo. Poichè il volume di controllo che si andrebbe ad introdurre è

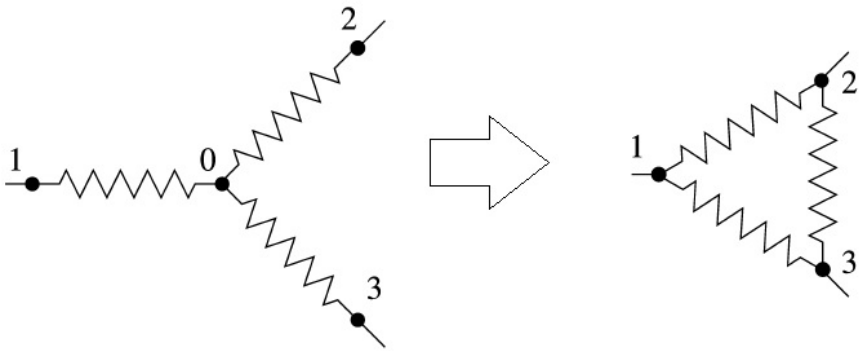



**Figura 2.4:** Analogia stella triangolo

molto più piccolo degli altri, i conti si semplificano e si arriva alla seguente formula per la componente geometrica della trasmissibilità

$$T_{ij} = \frac{\beta_i \beta_j}{\sum_{k=1}^{n} \beta_k}, \tag{2.10}$$

dove $n$ è il numero di fratture che si intersecano in quel punto o in quel segmento.

Per quanto riguarda le condizioni al contorno, quelle di tipo Neumann sono molto facili da implementare; se $\nabla p \cdot \mathbf{n} = g$ su $\Gamma \subset \partial\Omega$, il flusso numerico

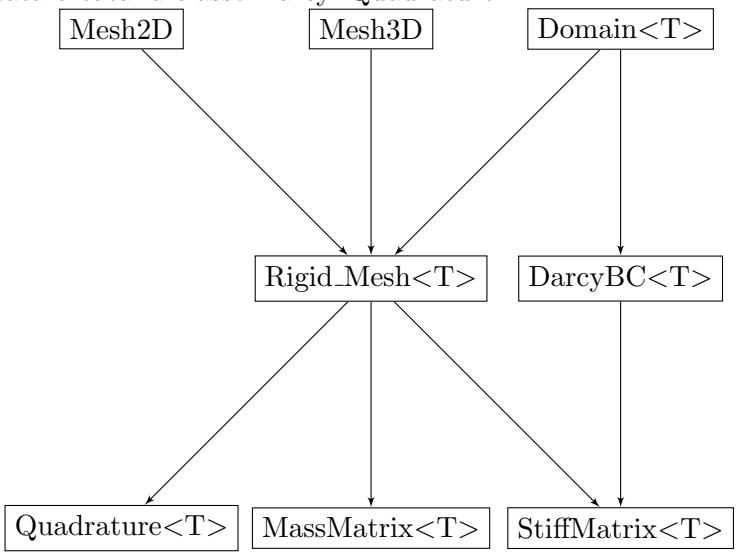
per quel lato può essere calcolato come la lunghezza del lato moltiplicata per un'opportuna discretizzazione di $g$. Più complicato è invece il caso delle condizioni al contorno di tipo Dirichlet. Il metodo più diffuso e implementato anche nel nostro codice, consiste nel creare una cella fittizia fuori dal dominio, subito adiacente al lato attraverso cui si vuole calcolare il flusso. In questa cella si impone il valore della soluzione dato dal valore al bordo di Dirichlet. Il flusso fra queste due celle viene quindi calcolato grazie a (2.8) e (2.9).

# Capitolo 3

# Il codice in C++

Per capire meglio la struttura del codice, diamo prima uno sguardo generale alle classi implementate. Partendo dalle considerazioni fatte nel capitolo precedente, sappiamo che la prima cosa da fare per risolvere l'equazioni di Darcy in un mezzo poroso con fratture è creare una suddivisione in volumi del dominio; questa suddivisione deve però tenere conto della presenza delle fratture nella maniera spiegata precedentemente. La classe Geometry::Mesh2D, implementata da L. Turconi, è in grado di svolgere questo lavoro egregiamente sfruttando la libreria CGAL per la generazione di griglie. Una volta generata, la griglia contenuta in Mesh2D può essere raffinata a piacimento e gli elementi possono essere modificati con grande facilità e velocità. Questa grande malleabilità della struttura, tuttavia, ha come svantaggio di rendere costoso l'accesso ai singoli elementi.
Il primo passo di questo progetto è stato quindi implementare una classe Geometry::Rigid_Mesh che potesse rendere veloce il riempimento di una matrice derivante dalla discretizzazione a volumi finitidi un problema, ma che, per contro, non permettesse nessun cambiamento della struttura. La classe Rigid_Mesh ha il grande vantaggio di essere compatibile anche con una struttura di tipo Mesh3D, ancora in via di implementazione.
Poiché nella classe Mesh2D non esiste alcun metodo per identificare a quale lato del dominio appartenga una facet di bordo, è stata implementata la classe Geometry::Domain, che contiene al suo interno i bordi del dominio etichettati. Nel suo costruttore, Rigid_Mesh etichetta le Facet appartenenti ad un lato del dominio in modo da permettere, in seguito, l'imposizione di condizioni di bordo.
Una volta creata la mesh, è stata implementata la classe base astratta Darcy::MatrixHandler da cui derivano le due classi Darcy::StiffMatrix e Darcy::MassMatrix, che costruiscono e contengono rispettivamente le matrici di stiffness e di massa per il problema di Darcy. Quest'ultima può tornare utile qualora si voglia risolvere l'equazione di Darcy nella sua versione tempo dipendente.

Per passare alla classe StiffMatrix le condizioni al contorno del problema, è stata implementata la classe Darcy::BoundaryConditions. Infine per calcolare la norma $L^2(\Omega)$ della soluzione e per discretizzare una forzante esterna è stata creata la classe Darcy::Quadrature.




Nel diagramma rappresentato, la classe indicata dalla freccia ha bisogno delle classi da cui essa parte per essere costruita.

## 3.1 La classe Dimension

Come è stato anticipato più volte l'obiettivo di questo progetto era creare un codice per risolvere l'equazione di Darcy sia in due che in tre dimensioni. Nonostante una griglia bidimensionale e una griglia tridimensionale abbiano tantissimi tratti in comune, esistono alcuni elementi in cui esse differiscono. Tuttavia, lasciare che le classi implementate scelgano al momento dell'esecuzione se usare un metodo per una griglia 2D oppure un metodo per una griglia 3D, può risultare molto inefficiente e computazionalmente costoso. Inoltre, in alcuni casi questo approcio può causare errori al momento della compilazione. La soluzione migliore è di decidere già a livello di compilatore quale metodo usare. Per implementare questo tipo di soluzione si rende necessaria l'introduzione di un type-trait.
I type-trait si basano su una semplice considerazione: gli enum vengono valutati al momento della compilazione. Nel nostro caso è stata implementata la classe Geometry::Dimension che ammette come parametro template un intero $N$. Questa classe è vuota. Tuttavia abbiamo creato due specializzazioni della classe una per il parametro template $N = 2$ ed una per $N = 3$.

```
template<int N>
struct Dimension
```

```
{};

template<>
struct Dimension<2>
{
typedef Geometry::Point2D Generic_Point;
typedef Geometry::Mesh2D::Edge2D Generic_Facet;
typedef Geometry::Mesh2D::Cell2D Generic_Cell;
typedef Geometry::Mesh2D Generic_Mesh;
//altri typedef
enum {dim=2};
};

template<>
struct Dimension<3>
{
typedef Geometry::Point3D Generic_Point;
typedef Geometry::Mesh3D::Facet3D Generic_Facet;
typedef Geometry::Mesh3D::Cell3D Generic_Cell;
typedef Geometry::Mesh3D Generic_Mesh;
//altri typedef
enum {dim=3};
};
```

Definite queste specializzazioni possiamo passare, per esempio, alla classe Rigid_Mesh<T> come parametro template Dimension<2> o Dimension<3>. A questo punto chiamando

```
showDimension ( T () ) ;
```

viene scelto al momento della compilazione se verrà chiamato quello relativo al caso bidimensionale o quello relativo al caso tridimensionale:

```
void showDimension ( const Dimension<2>,
std::ostream & out=std::cout ) const;
void showDimension ( const Dimension<3>,
 std::ostream & out=std::cout ) const;
```

Un'altra importante applicazione della classe Dimension<N> è dovuta ai typedef che essa contiene. Il costruttore della classe Rigid_Mesh<T>, ad esempio, ha bisogno che una referenza ad una Mesh2D o ad una Mesh3D venga ad esso passata come variabile. Risulterebbe scomodo, tuttavia, implementare due costruttori differenti. Grazie ai typedef contenuti in Dimension<N> e

alla parola typename, la classe Rigid_Mesh a questo punto è in grado di trattare sia una griglia Mesh2D che una griglia Mesh3D come una Generic_Mesh anche se sono due oggetti diversi.

```
template <class T>
class Rigid_Mesh{
public:
typedef typename T::Generic_Facet Generic_Facet;
typedef typename T::Generic_Point Generic_Point;
typedef typename T::Generic_Mesh Generic_Mesh;
typedef typename T::Generic_Cell Generic_Cell;
//altri typedef
Rigid_Mesh (Generic_Mesh & generic_mesh, Domain<T>& domain);
//altri membri
}
```

Grazie a questo tipo di soluzione è quindi possibile chiamare lo stesso costruttore sia con una Mesh2D che con una Mesh3D e le varie scelte vengono fatte tutte a livello di compilazione. Questo tipo di soluzione permette quindi di diminuire il codice da implementare, e, soprattutto rende il codice molto efficiente.

## 3.2 La classe Rigid_Mesh

La classe Rigid_Mesh è il cuore di questo progetto: una sua implementazione efficiente e incentrata sulle applicazioni ha reso molto più semplice ed efficiente la programmazione delle classi che da essa dipendono, ossia StiffMatrix, MassMatrix e Quadrature. Come è già stato parzialmente anticipato, questa classe può contenere sia una griglia bi- che una griglia tridimensionale. Le celle delle griglie possono avere un numero arbitrario di facce, l'unica restrizione è che siano poligonali o poliedriche. Se il caso trattato è in 2D o in 3D viene comunicato a Rigid_Mesh passando come parametro template, rispettivamente, il tipo Dimension<2> o il tipo Dimension<3> come mostrato nella sezione precedente.

La prima scelta che è stata fatta per quanto riguarda l'implementazione è di usare un approcio di tipo nested-classes ossia di classi innestate (si veda ad esempio [2]). All'interno della classe Rigid_Mesh, sono state infatti implementate altre classi quali Cell e Facet, che servono a contenere le facce e le celle della mesh e che, difficilmente, hanno un utilizzo al di fuori della classe per la griglia. Presentiamo innanzitutto la struttura generale della griglia:

```
template <class T>
class Rigid_Mesh{
```

```
public:

//typdef typename di tipi definiti dento il parametro template T

class Facet{};
class Cell{};
class Facet_ID{};
class Regular_Facet : public Facet_ID {};
class Border_Facet: public Facet_ID {};
class Fracture_Facet: public Facet_ID {};

public:
//Costruttori
//metodi di tipo get
//altri metodi
protected:
//metodi protetti che servono al costruttore o ad altri metodi

protected:
std::vector<Point> M_nodes;
std::vector<Facet> M_facets;
std::vector<Cell> M_cells;

std::vector<Regular_Facet> M_internalFacets;
std::vector<Border_Facet> M_borderFacets;
std::vector<Fracture_Facet> M_fractureFacets;

Domain<T> M_domain;
};
```

Nonostante la classe Rigid_Mesh nasca per risolvere il problema di Darcy in mezzi porosi con fratture, la struttura è stata disegnata in modo tale da renderla fruibile in altri contesti. A tal proposito si è scelto di non differenziare le Facet di frattura da quelle non di frattura in modo tale che questa classe possa essere utilizzata anche per problemi differenziali diversi da quello per cui nasce. Questa griglia, inoltre, conserva una struttura abbastanza generale che permette di sviluppare in futuro un solutore a differenze mimetiche per il problema di Darcy in mezzi con fratture.

Come è stato mostrato nel capitolo precedente, per costruire la matrice di stiffness per un problema discretizzato con volumi finiti, bisogna valutare il flusso attraverso ogni faccia del dominio. Per non dover calcolare il flusso attraverso ogni Facet due volte, ho deciso di iterare su tutte le facce del dominio e calcolare il flusso una volta sola. Questa scelta ha comportato

importanti ricadute sulla struttura di Rigid_Mesh. La prima conseguenza è che la classe Facet risulta molto ricca di informazioni mentre la classe Cell risulta molto leggera. La seconda è la creazione dei vettori M_internalFacets, M_borderFacets e M_fractureFacets che contengono al loro interno gli Id delle facce regolari, di bordo e di frattura rispettivamente. Al momento di costruire la matrice di stiffness questa scelta evita al calcolatore di dover mettere degli if, che sono computazionalmente costosi, per capire se una faccia è di bordo o di frattura o nessuna delle precedenti.

Le strutture di dati utilizzate nella parte protected sono tutte std::vector. Questa scelta è stata fatta perché i vector sono strutture che consentono un accesso agli elementi poco costoso e sono facili da utilizzare. Non dovendo fare ricerche di elementi, che spesso portano a privilegiare altre strutture, o non dovendo ordinare gli elementi, gli std::vector sono sicuramente una scelta ottimale.
Prima di passare ad esaminare più approfonditamente i membri della classe e le classi innestate ripetiamo che, poiché il cambiamento, l'aggiunta o l'eliminazione di un nodo, di una cella o di una faccia dalla griglia sarebbe computazionalmente molto costosa con la struttura adottata, si è scelto di non consentire all'utente nessuna modifica. Questa scelta implica che non esistono metodi pubblici della classe Rigid_Mesh, né delle classi ad essa innestate, che permettono il cambiamento dei membri private o protected in esse contenuti.

Passiamo ora in rassegna le varie classi contenute in Rigid_Mesh.

### 3.2.1 La classe Facet

La struttura generale di questa classe è la seguente:

```
class Facet{

public:
Facet (const Generic_Facet & generic_facet,
  Geometry::Rigid_Mesh<T> * const mesh,
     const std::map<UInt, UInt> &old_to_new_map,
    const UInt m_id);
Facet (const Facet& facet, Geometry::Rigid_Mesh<T> *const mesh);
Facet (const Facet& facet);
Facet () = delete;
//metodi di tipo get

protected:
Geometry::Rigid_Mesh<T> * M_mesh;
```

```
UInt M_Id;
std::vector<UInt> M_Vertexes_Ids;
std::vector<UInt> M_separatedCells_Ids;
Real M_size;
Generic_Point M_center;
Generic_Vector M_UnsignedNormal;

protected:
//metodi di tipo protetto
};
```

Per prima cosa si osservi che il costruttore vuoto è stato eliminato; questa scelta viene ripetuta in tutte le altre classi ed è per non avere metodi esterni che possano cambiare i membri protected. Il costruttore più importante è il primo ed è quello che viene chiamato nel costruttore della Rigid_Mesh; in questo costruttore viene data una faccia proveniente da una Mesh2D o da una Mesh3D dalla quale vengono prese le informazioni sul lato e un puntatore alla Rigid_Mesh che lo contiene, visto che le nested classes non possono accedere direttamente agli elementi della classe che le contiene. L'ultimo argomento che contiene il costruttore è una mappa da intero a intero. Il suo utilizzo è il seguente: nella Mesh2D o nella Mesh3D alcune celle possono essere state cancellate volutamente dall'utente e quindi può mancare la cella corrispondente a un certo indice. In Rigid_Mesh invece, se le celle sono $N$, gli indici delle celle vanno da 0 a $N-1$. Questa mappa serve quindi al momento della costruzione a passare dagli indici delle celle che confinano attraverso quella faccia vecchi a quelli nuovi. Si noti inoltre che esistono due costruttori di copia: il primo viene chiamato dal costruttore di copia della classe Rigid_Mesh e quindi permette al nuovo oggetto di avere un puntatore all'oggetto a cui appartiene e non a quello che è stato copiato.

I metodi di tipo *get* sono quelli che permettono all'utente di poter vedere le variabili che sono protected. I metodi di tipo protetto sono metodi chiamati dal costruttore per calcolare la normale alla faccia e il centroide. Si noti che il metodo per calcolare la normale cambia a seconda che si sia in 2D o 3D. È importante notare che tutto ciò che è necessario per calcolare il flusso numerico attraverso (2.8) e (2.9) è contenuto in questa classe; l'unico elemento mancante è il centroide delle celle confinanti, che può essere facilmente recuperato grazie agli indici delle celle separate e al puntatore alla classe madre. Si noti che il prodotto scalare tra $\mathbf{f_i}$ e $\mathbf{n_i}$ in (2.9) è sempre positivo. Non importa quindi se la normale $\mathbf{n}$ è entrante o uscente, ma è sufficiente prendere il valore assoluto del prodotto scalare.

### 3.2.2 La classe Cell

La struttura generale della classe Cell è per molti versi simile a quella della classe Facet: riportiamo qualche stralcio di codice per illustrarne la struttura generale.

```
class Cell{

public:
Cell (const Generic_Cell & generic_cell,
 Geometry::Rigid_Mesh<T> *const mesh, const UInt m_id);
Cell (const Cell& cell,  Geometry::Rigid_Mesh<T> * const mesh);
Cell (const Cell& cell);
Cell () = delete;

//metodi di tipo get
bool hasNeighborsThroughFacet (const UInt & facetId,
    const UInt & idNeighbor) const;
friend class Rigid_Mesh<T>;

protected:
Geometry::Rigid_Mesh<T> * M_mesh;
UInt M_Id;
std::vector<UInt> M_Vertexes_Ids;
std::vector<UInt> M_Neighbors_Ids;
Generic_Point M_centroid;
Real M_volume;

protected:
//metodi tipo protected
};
```

Per la classe Cell possiamo ripetere le considerazioni fatte sui costruttori già per la classe Facet: il costruttore principale prende una Cell2D o una Cell3D da cui prende le informazioni della cella, il proprio Id e il puntatore alla classe che lo contiene. Come nel caso delle Facet esiste una versione del costruttore di copia chiamata dal costruttore di copia della Rigid_Mesh e una versione standard. Infine il costruttore senza argomenti è stato cancellato.

Come è stato spiegato poco sopra, in una Mesh2D o in una Mesh3D gli indici delle celle possono differire, a causa di cancellazioni, dagli indici delle celle utilizzati nella Rigid_Mesh. A tal proposito il costruttore di quest'ultima, dopo aver ridato un indice ad ogni cella, deve correggere il vettore M_Neighbours_Ids contenente gli indici delle celle confinanti alla Cell in esa-

me. Come spiegato più volte, però, non si è voluto implementare una funzione che potesse modificare i membri della classe a meno di non renderli protetti. A tal proposito si è quindi deciso di rendere la classe Rigid_Mesh *friend* della classe Cell in modo tale da poter fare questa correzione. Si noti che questa proprietà non è fornita dal semplice innestamento delle classi(si veda a tal proposito [2]).

Nella classe Cell è stato anche implementato un metodo che consente di determinare se la cella Cell ha come vicino una faccia di indice idNeighbor attraverso la faccia di indice facetId.

### 3.2.3 La classe Facet_Id e le classi derivate

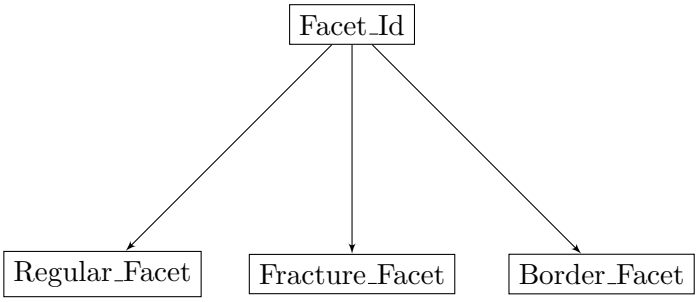
Come spiegato in precedenza, per mantenere una struttura della classe Rigid_Mesh che potesse rendere questa classe utile anche in applicazioni diverse da quella per cui è stata disegnata, si è creato un vettore di Facet tutte uguali. Tuttavia, soprattutto nella nostra applicazione, non tutte le facce sono equipollenti, in quanto alcune rappresentano fratture, altre lati di bordo del dominio, altre né le une né le altre. Per avere accesso alle facce dei rispettivi tipi, sono stati creati tre vettori ognuno con gli indici delle facce di un tipo diverso: le facce di bordo, le facce di frattura e le facce regolari, che non sono né di bordo né di frattura. Tuttavia nei vettori gli indici non sono salvati come interi, ma sono contenuti all'interno di una piccola classe, che, nel caso di faccia di frattura o di bordo contiene anche ulteriori informazioni. Le varie classi sono tutte derivate dalla classe Facet_Id che contiene al suo interno un ID di una Facet e un puntatore alla Rigid_Mesh che la contiene.

```
class Facet_ID{

protected:
UInt Facet_Id;
Rigid_Mesh<T>* M_mesh;

public:
Facet_ID (const UInt facet_id,  Geometry::Rigid_Mesh<T> *const mesh);
Facet_ID () = delete;
virtual ~Facet_ID () {};
//metodi di tipo get
};
```

I metodi di tipo get danno l'accesso diretto alla Facet e ai suoi principali metodi, ma a causa dei vari reindirizzamenti sono una scelta comoda ma non molto efficiente.




Come mostrato dallo schema sono state implementate tre classi che derivano da Facet_Id: Regular_Facet, Fracture_Facet e Border_Facet. La prima di queste serve a contenere gli indici delle celle che non sono né di frattura né sul bordo del dominio ed è , di fatto, uguale alla classe base.
La classe Border_Facet contiene invece qualche leggera modifica al suo interno; oltre all'indice di una Facet e al puntatore alla Rigid_Mesh ereditati da Facet_Id, essa contiene l'indice del bordo del dominio a cui appartiene. Questo indice è quello che viene dato a quel bordo del dominio nella classe Domain alla quale, in Rigid_Mesh, c'è una referenza. Questo indice viene passato al costruttore di Border_Facet dal costruttore della classe che lo contiene.
Infine dalla classe Facet_Id deriva la classe Fracture_facet, fra le classi derivate, di certo la più complessa. Questa classe tiene al suo interno, oltre che l'indice di una Facet di frattura, le informazioni della frattura che questa faccia rappresenta.

```
class Fracture_Facet: public Facet_ID {
public:
//costruttore
//costruttori di tipo copy constructor
//metodi di tipo get

friend class Rigid_Mesh<T>;

private:
UInt Cells_number;
UInt M_Id;
Real M_permeability;
Real M_aperture;
std::vector<UInt> Fracture_Ids;
std::map<Fracture_Juncture, std::vector<UInt> > Fracture_Neighbors;

};
```

Come si può notare dal codice, questa classe contiene al suo interno anche un proprio Id in quanto Fracture_Facet: questo Id viene sommato al numero di celle nella classe della mesh, in modo tale da fornire un Id della frattura in quanto cella. Poiché una Facet può rappresentare piú di una frattura contemporaneamente, si è calcolata un'apertura totale della frattura sommando le varie aperture rappresentate e una permeabilità totale facendo una media pesata delle permeabilità rappresentate.

La mappa Fracture_Neighbors associa ad ogni Fracture_Juncture un vettore di indici di facce di frattura. Una Fracture_Juncture è un punto in 2D e un segmento in 3D (più precisamente un indice ad un nodo in 2D ed un pair di indici di nodi in 3D) e rappresenta la giunzione fra due facce frattura. La mappa associa quindi ad ogni giunzione gli indici delle fratture con cui la classe comunica attraverso essa. Questa mappa permette in maniera rapida di assemblare la matrice grazie alla formula (2.10). Poiché questa mappa va creata dopo aver creato il vettore di Fracture_Facet contenuto in Rigid_Mesh, quest'ultima è stata dichiarata come classe friend. Nel caso tridimensionale Gli indici dei nodi vengono ordinati in modo da identificare in maniera univoca la giuntura.

È importante notare che, anche se le classi derivate sono fra loro diverse, poiché derivano tutte da Facet_Id è possibile usare il polimorfismo. Inoltre, in questo lavoro è stato ipotizzato che una faccia non possa essere di bordo e di frattura contemporaneamente. Nel caso in cui un problema dovesse presentare questa peculiarità, il codice, con la sua struttura, è facilmente generalizzabile in modo da poter risolvere questo problema.

### 3.2.4 Metodi di Rigid_Mesh

I metodi della classe Rigid_Mesh si possono dividere in:

- Costruttori;
- Metodi di tipo get;
- Altri metodi;
- Metodi protetti.

Come i costruttori delle classi innestate, il costruttore senza argomenti è stato eliminato, ma è stato implementato il costruttore di copia. Il costruttore di Rigid_Mesh prende come argomenti una Mesh2D o una Mesh3D dalle quali copia le informazioni sulla griglia e un dominio di tipo Geometry::Domain<T> che contiene le informazioni necessarie per dire alle Border_Facet a quale bordo appartiene.

```
Rigid_Mesh (Generic_Mesh & generic_mesh, Domain$<$T$>$& domain);
```

I metodi di tipo get servono per avere accesso in lettura alle informazioni salvate nella parte protected della classe e quindi ai vettori contenenti i nodi, le facce, le celle, e le tre classi derivate da Facet_Id. Oltre a questi metodi ci sono dei metodi che permettono di sapere qual è la dimensione della Facet più grande, quella della più piccola e la dimensione media delle Facet.

Fra gli altri metodi c'è un metodo che serve per salvare la griglia in formato vtk e uno che serve per appendere a un file una funzione assegnata per punti o per celle. La prospettiva è quella di salvare in un file la griglia usando la prima di queste due funzioni, e poi appendere all'interno del file la soluzione di un problema differenziale ottenuta con una qualsiasi discretizzazione, usando il secondo. Quest'ultimo ha fra gli argomenti *label*, che è il nome con cui si vuole etichettare la variabile da salvare, e *solType*: quest'ultimo è molto importante perché permette di salvare la soluzione per punti o per celle e consentendo quindi il salvataggio di soluzioni calcolate con metodi numerici che calcolano la soluzione non sulle celle ma sui nodi della griglia.

```
bool exportVtk(const std::string & fileName) const;
bool appendSolutionToVtk (Darcy::Vector& sol,
  const std::string & fileName,
   const std::string & label = "pressure",
     const std::string & solType = "CELL") const;
```

Oltre a questi metodi la classe Rigid_Mesh ha un metodo *showme* che mostra le principali caratteristiche della griglia.

È importante sottolineare che, fra i vari vantaggi di salvare la soluzione in formato vtk, vi è anche quello di poter salvare i lati frattura come se fossero delle celle. Infatti, ad esempio, in una mesh bidimensionale è possibile salvare come celle anche degli oggetti monodimensionali. Questa possibilità ci permette di salvare la soluzione anche all'interno delle fratture e di poterla poi plottare con Paraview, senza dover apportare cambiamenti alla griglia.

Infine abbiamo i metodi protetti: questi ultimi sono metodi chiamati dal costruttore o da altri metodi che sono diversi nel caso 2D dal caso 3D e pertanto svolgono delle piccole parti di lavoro che per forza devono essere implementate diversamente.

## 3.3 Le classi Domain e BoundaryConditions

Le classi Domain<T> e la classe BoundaryConditions<T> sono state entrambe create per imporre delle condizioni al contorno ad un problema differenziale che si vuole risolvere in questo dominio. Questo tipo di oggetti

funziona per qualsiasi tipo di dominio 2D o 3D a patto che sia poligonale o poliedrico.

### 3.3.1 La classe Domain

La classe Domain è una classe che contiene i bordi del dominio su cui è stata generata la griglia. La sua struttura è molto semplice, basata ancora sulle classi annidate. Il namespace a cui appartiene è ancora il namespace Geometry, come Rigid_Mesh, Mesh2D e Mesh3D.

```
template <class T>
class Domain
{
typedef typename T::Generic_Point Generic_Point;
typedef typename T::Generic_Segment Generic_Segment;
typedef typename T::Generic_Border Generic_Border;

public:
class DomainBorder {
protected:
UInt m_Id;
Generic_Border borderFacet;
public:
//costruttore e costruttore di copia
//metodi di tipo get
private:
//metodi private
};

const UInt getBorderId(const std::vector<Point2D> facetVertexes) const;
const UInt getBorderId(const std::vector<Point3D> facetVertexes) const;
//altri metodi di tipo get

Domain(const std::vector<DomainBorder>& border);
Domain(const Domain<T> & domain) = default;
~Domain() = default;

protected:
std::vector<DomainBorder> BordersVector;
};
```

Come è facile vedere dal codice, la classe Domain si costruisce prendendo un vettore di bordi. La caratteristica più importante di questi oggetti è che hanno il metodo getBorderId che, sfruttando metodi della libreria CGAL, sono in grado di dire su quale lato di bordo si trova la faccia. Come nei casi

precedenti, i metodi protected sono metodi chiamati dal costruttore e che eseguono una parte di lavoro che differisce fra il caso bidimensionale e quello tridimensionale.

È importante osservare che, in alcuni casi, si vuole imporre due tipi diversi di condizioni al contorno sullo stesso lato di bordo. In tal caso è sufficiente dividere in due il lato di bordo e passare a Domain le due parti come se fossero due bordi diversi.

### 3.3.2 La classe BoundaryConditions

La classe BoundaryConditions è una classe che contiene le condizioni al contorno che si vogliono imporre per un dato problema. Poiché questi oggetti contengono al proprio interno una referenza ad un oggetto di tipo Domain<T> che è una classe template, anche questa classe deve essere una classe template. Differentemente dalle classi precedenti, però, questa è stata implementata nel namespace Darcy invece che in quello Geometry. Questo namespace contiene al suo interno tutti i tool implementati, che sono necessari per risolvere problemi differenziali su una Rigid_Mesh.

Le condizioni al contorno che sono state implementate sono di tipo Dirichlet o Neumann; grazie ad una *enum* si è provveduto a distinguere i due casi:

```
enum BCType{Dirichlet, Neumann};
```

La sua struttura di BoundaryConditions è molto simile a quella di Domain:

```
template <class T>
class BoundaryConditions
{
//typedef necessari
public:
class BorderBC {
protected:
UInt m_Id;
BCType bcType;
std::function<D_Real(Generic_Point)> BC;
BoundaryConditions<T>* m_bcContainer;
public:
BorderBC (UInt Id, BCType bctype,
 std::function<D_Real(Generic_Point)> bc);
//metodi di tipo get e costruttore di copia
};
//metodi di tipo get
```

```
BoundaryConditions (std::vector<BorderBC>& borderbc,
  Geometry::Domain<T>& domain);
BoundaryConditions (const BoundaryConditions&) = default;

protected:
std::vector<BorderBC> BordersBCVector;
Geometry::Domain<T>& M_domain;
};
```

Si noti innanzitutto la struttura annidata come in precedenza; ogni BorderBC contiene al suo interno un Id di un lato di bordo e una funzione che descrive la condizione al contorno da imporre. Quest'ultimo passaggio viene fatto grazie alle std::function, un elemento introdotto nel nuovo standard C++11. Come anticipato prima, ogni BorderBC ha al suo interno una variabile di tipo BCType che permette di specificare se la condizione al contorno è di tipo Neumann o di tipo Dirichlet. Il costruttore di BoundaryConditions si premura di mettere in ordine gli elementi di BordersBCVector per indice del bordo e a verificare che non ci siano due condizioni al contorno sullo stesso lato. Riportiamo qui lo stralcio di codice in cui il vettore viene ordinato:

```
auto cmp = [](BorderBC bc1, BorderBC bc2)
{return (bc1.getId()<bc2.getId());};
std::sort (BordersBCVector.begin(),BordersBCVector.end(), cmp);
```

Si noti che viene usata una lambda-function, novità dello standard C++11 e che viene usato l'algoritmo sort della standard library. La sintassi usata è molto efficiente.

## 3.4 La classe MatrixHandler

La classe MatrixHandler è una classe base di tipo astratto, ossia una classe base che contiene al suo interno un metodo di tipo *pure virtual*. Ricordiamo che gli oggetti di questo tipo non possono essere costruiti, ma solo basi da cui ereditare. Gli oggetti derivanti da MatrixHandler assemblano una matrice derivante dalla discretizzazione di un problema differenziale su di una griglia contenuta in una Rigid_Mesh. Poiché quest'ultima è una classe template, anche MatrixHandler lo sarà. Osserviamone innanizutto la struttra:

```
enum DiscretizationType {D_Cell=0, D_Nodes=1};

template <class T>
class MatrixHandler {

typedef DiscretizationType DType;
```

```
public:
MatrixHandler(const Geometry::Rigid_Mesh<T> &rigid_mesh, DType dtype=D_Cell);
MatrixHandler(const MatrixHandler&) = delete;
MatrixHandler() = delete;
virtual ~MatrixHandler() {};

SpMat& getMatrix() const
{return *M_Matrix;}
UInt getSize() const
{return M_size;}
virtual void assemble()=0;

protected:
const Geometry::Rigid_Mesh<T> & M_mesh;
DType M_policy;
D_UInt M_size;
std::unique_ptr<SpMat> M_Matrix;
};
```

La prima cosa che è stata fatta è lasciare la possibilità all'utente di decidere se la matrice deve avere la dimensione del numero di nodi della griglia, o del numero di celle. Si è deciso di non passare questo parametro come un parametro template perché il dimensionamento viene comunque fatto al momento dell'esecuzione. Come si può osservare, l'utente può specificare questa opzione al costruttore: di default la dimensione è quella del numero di celle. Il costruttore poi sfrutta il fatto che ad ogni enum corrisponde un intero: infatti

```
 M_size ((1-dtype) *(rigid_mesh.getCellsVector().size()
+rigid_mesh.getFractureFacetsIdsVector().size())
+dtype*(rigid_mesh.getNodesVector().size()))
```

e quindi nel caso si scelga D_Cell, (1-dtype)= 1 e dtype= 0, nel caso invece si scelga D_Nodes abbiamo che(1-dtype)= 0 e dtype= 1. Nel costruttore si può quindi costruire la matrice con

```
 M_Matrix(new SpMat(this->M_size, this->M_size))
```

perché M_size è già stato definito.

Il metodo assemble, qui dichiarato puramente virtuale, è quello che viene sovrascritto dalle varie classi ereditate ed è il metodo che assembla la matrice. Il costruttore di copia è stato cancellato perchè ha poco senso copiare tutta la classe: è preferibile creare solo una nuova matrice copiando

la vecchia. Il distruttore, come in ogni classe base, è stato dichiarato virtual.

Si noti a questo punto che la matrice M_Matrix è tenuta nella classe tramite un puntatore. Questo fatto è una conseguenza diretta della scelta che è stata fatta per la libreria di algebra lineare. Infatti, ricordiamo che SpMat non è altro che

```
Eigen::SparseMatrix<D_Real>
```

e quindi una matrice di tipo sparso della libreria Eigen. Quest'ultima è stata implementata sfruttando tecniche di programmazione avanzata ed è quindi molto efficiente. Tuttavia queste matrici sparse non possono essere tenute in una classe a meno che questa non occupi un preciso numero di byte o un suo multiplo. Per non rinunciare all'efficienza delle Eigen si è quindi scelto di aggirare questa difficoltà tenendo la matrice nella classe solo grazie ad un std::unique_pointer. Quest'ultimo è uno smart-pointer introdotto nell'ultimo standard, molto pratico da usare perché ha il grande vantaggio di liberare da solo la memoria da lui puntata.

## 3.5 Le classi StiffMatrix e MassMatrix

Queste due classi derivano entrambe dalla classe MatrixHandler e si occupano, rispettivamente, di costruire la matrice di stiffness e la matrice di massa per un problema differenziale discretizzato con volumi finiti. StiffMatrix e MassMatrix nascono per risolvere il problema di Darcy in mezzi porosi con fratture, sia in due dimensioni che in tre dimnesioni. Nonostante questa origine, i problemi a cui possono essere applicate sono molteplici.

Come la loro classe base queste due classi sono di tipo template, in quanto contengono al loro interno una referenza ad una Rigid_Mesh che è una classe template. Il parametro template che quindi si può passare è di tipo Dimension<2> nel caso bidimensionale e di tipo Dimansion<3> nel caso tridimensionale. Come MatrixHandler, sia MassMatrix che StiffMatrix si trovano nel namespace Darcy.

### 3.5.1 MassMatrix

La matrice di massa può servire qualora si voglia calcolare l'avanzamento temporale dell'equazione di Darcy, o, più semplicemente, qualora la PDE in esame presenti un termine di reazione con coefficiente costante.
MassMatrix non presenta diversità dalla classe base da cui deriva, se non per il metodo virtual assemble che è stato implementato. Nel caso in esame assemble() itera su tutte le celle della mesh e mette sulla componente i-esima della diagonale il volume della cella i-esima. Finito questo ciclo si itera sulle

Fracture_Facet e si calcola il volume della cella rappresentata come la lunghezza (o l'area in 3D) della Facet moltiplicata per l'apertura della cella rappresentata. Come anticipato precedentemente ogni faccia di frattura ha un Id in quanto cella e questo viene usato per riempire la matrice di massa con il volume anche delle celle di frattura.

Poiché tenendo conto del volume delle fratture il volume totale del dominio aumenta, è stata implementata una correzione al volume delle celle vicino alla facce di frattura. In questo modo la somma degli elementi sulla diagonale della matrice di massa sono uguali al volume totale del dominio.

### 3.5.2 StiffMatrix

La classe SiffMatrix implementa la matrice di stiffness secondo l'algoritmo descritto nel capitolo precedente. In quest'ultimo, in realtà, è già stato mostrato che il metodo si può applicare a problemi di tipo diverso e in casi senza fratture. Come anticipato precedentemente questa classe deriva da MatrixHandler.

Il costruttore di questa classe è il seguente:

```
StiffMatrix(const Geometry::Rigid_Mesh<T> &rigid_mesh,
 BoundaryConditions<T>& Bc,
   std::function<D_Real(Generic_Point)> permeability,
      D_Real mobility = 1.);
```

Come prevedibile questo costruttore ha bisogno di molte più variabili rispetto a quello di MassMatrix. Differentemente da prima, per costruire una matrice di rigidezza per il problema in esame abbiamo bisogno di condizioni al contorno. Queste vengono passate attraverso una classe BoundaryConditions. Per evitare problemi all'utente dummy si è scelto di dover imporre una condizione al contorno su ogni lato, non lasciandone nessuna implicita. Il secondo parametro di cui ha bisogno il costruttore è una funzione, passata attraverso una std::function, che rappresenta la permeabilità all'interno del dominio. Volendo, è possibile passare alla classe in esame, anche la mobilità del fluido. Nel caso non si risolva il problema di Darcy ma altri problemi, questa può essere lasciata al suo valore di default che è uno. In tal caso la funzione permeability fa le veci della funzione $K$ utilizzata nel precedente capitolo o di un generico coefficiente di diffusione isotropo ma non omogeneo.

Oltre alle variabili protected, ereditate da MatrixHandler, in StiffMatrix vi sono le seguenti variabili alle quali non si può accedere dall'esterno:

```
std::unique_ptr<Vector> _b;
std::function<D_Real(Generic_Point)> M_Permeability;
```

```
D_Real M_Mobility;
BoundaryConditions<T>& m_Bc;
```

Oltre alle variabili che già sono state spiegate, in questa classe abbiamo il vettore _b. Questo vettore è un vettore della libreria Eigen, ed è il vettore termine noto contenente le condizioni al contorno per il problema.

Il metodo assemble, che dal punto di vista teorico non è di semplice implementazione, ha una struttura relativamente semplice per merito di come è stata costrutia la classe Rigid_Mesh. La struttura è la seguente:

1. Si itera sulle facce che non sono né di frattura né di bordo con un ciclo for sul vettore delle Regular_Facet e, ad ogni iterazione,
   (a) si calcolano i coefficienti $\beta_i$ dati dalla formula (2.9);
   (b) si calcola $T_{ij}$ con la formula (2.8) e si calcola il flusso numerico $H_{ij}$;
   (c) si aggiunge alla matrice nelle posizioni $(i, i)$, $(j, j)$, $(i, j)$ e $(j, i)$ il flusso numerico calcolato con il segno corretto.
2. Si itera sulle facce di frattura con un ciclo for sul vettore delle Fracture_Facet e, ad ogni iterazione,
   (a) si calcolano i coefficienti $\beta_i$ dati dalla formula (2.9), considerando la faccia di frattura come una cella. Le celle con cui essa confina sono due e quindi i $\beta_i$ da calcolare sono tre invece che due;
   (b) si calcolano i due $T_{ij}$ con la formula (2.8) e si calcolano i flussi numerici $H_{ij}$;
   (c) attribuito alla faccia di frattura l'indice come cella $k$ si aggiunge alla matrice nelle posizioni $(i, i)$, $(j, j)$, $(k, k)$, $(i, k)$, $(k, i)$, $(j, k)$ e $(k, j)$ il giusto flusso numerico calcolato con il segno corretto;
   (d) si itera sulla mappa Fracture_Neighbors e si calcolano i flussi fra la cella presente e le celle-frattura confinanti grazie alla formula (2.10);
   (e) si riempie la matrice ma non in maniera simmetrica, solo gli elementi sulla i-esima riga perché dobbiamo iterare su tutte le facce.
3. Si itera sulle facce di bordo con un ciclo for sul vettore delle Border_Facet e, ad ogni iterazione, si guarda se la condizione al contorno su quel lato è di tipo Dirichlet o Neumann:
   - se è di tipo Neumann la condizione viene mediata sul lato e moltiplicata per la misura di quest'ultimo e aggiunta al vettore _b;

- se è di tipo Dirichlet, viene creato un nodo fittizio fuori dal dominio al quale viene dato il valore della soluzione imposta dalle condizioni al contorno di Dirichlet. A questo punto si calcola il flusso con le formule (2.9) e (2.8) e si riempie la matrice ed il vettore _b del termine noto.

4. La libreria Eigen assembla la matrice in modo tale da rendere tutte le operazioni di algebra lineare rapide ed efficienti.

È importante notare che l'unico if che compare in tutta la procedura di assemblaggio, viene usato per verificare se un lato ha condizioni al contorno di tipo Neumann o di tipo Dirichlet. Nessun altro if, che computazionalmente sono costosi, compare all'interno della funzione assemble().

## 3.6 Le classi Quadrature e QuadratureRule

Queste due classi sono state create per integrare sul dominio di una Rigid_Mesh. La classe Quadrature ha la seguente struttura:

```
template <class T>
class Quadrature {
//alcuni typedef
public:
Quadrature (const Geometry::Rigid_Mesh<T> &rigid_mesh);
Quadrature (const Geometry::Rigid_Mesh<T> &rigid_mesh,
   const QuadratureRule<T>& quadrature);
Quadrature (const Geometry::Rigid_Mesh<T> &rigid_mesh,
  const QuadratureRule<T>& quadrature,
    const QuadratureRule<T>& fracturequadrature);

D_Real Integrate (const std::function<D_Real(Generic_Point)>& Integrand);
Vector CellIntegrate (const std::function<D_Real(Generic_Point)>& func);
D_Real Integrate (const Vector& Integrand);
D_Real L2Norm (const Vector& Integrand);

protected:
const Geometry::Rigid_Mesh<T> & M_mesh;
D_UInt M_size;
QR_Handler M_quadrature;
QR_Handler M_fractureQuadrature;
};
```

Di base, un oggetto di questo tipo, è in grado di fare due cose: integrare una funzione costante su ogni cella, e che quindi viene passata attraverso un vettore della libreria Eigen, e, data una funzione definita sul continuo,

restituire un vettore con l'integrale della funzione su ogni cella.

Per integrare una funzione costante sulle celle, quale può essere la soluzione di un problema differenziale discretizzato con i volumi finiti, è sufficiente chiamare la funzione Integrate(v) dove v è un vettore della libreria Eigen. Come è facile intuire questa funzione somma gli elementi del vettore moltiplicati per l'area dei rispettivi volumi.
La funzione L2Norm svolge lo stesso compito ma elevando al quadrato gli elementi del vettore e facendo la radice quadrata dell'integrale ottenuto. Si sottolinea che l'elevamento al quadrato è stato fatto moltiplicando l'elemento per se stesso, in quanto meno costoso computazionalmente rispetto al chiamre l'elevamento a potenza.
Poiché le fratture vengono rappresentate con spessore nullo, anche se tale non è, in entrambe le funzioni descritte sopra la funzione viene integrata anche su di esse. Tuttavia viene apportata una correzione all'integrale totale, in modo tale da conservare il volume totale del dominio.

La seconda funzionalità della classe in esame è svolta dalla funzione CellIntegrate. Questa funzione utilizza la formula di quadratura di tipo QuadratureRule che viene passata dal costruttore, per integrare la funzione su ogni cella e restituire un vettore della libreria Eigen con in posizione i-esima l'integrale della funzione sulla cella i-esima. Nel caso di dominio con fratture è possibile passare una regola di quardatura di tipo QuadratureRule per integrare la funzione integranda su quest'ultime. CellIntegrate è molto utile qualora si voglia discretizzare la forzante di un problema differenziale mediante il metodo dei volumi finiti. Nel caso venisse chiamato il costruttore di Quadrature senza specificare alcuna regola di quadratura, viene uitlizzata la regola del punto medio.
Come semplice corollario di questa funzione vi è la funzione Integrate che, chiamata con una std::function come argomento, restituisce l'integrale di quest'ultima.

Come si è già accennato, la classe QuadratureRule è una classe base astratta. Questa classe ha al suo interno un metodo clone() ed un metodo apply. La firma di quest'ultimo è la seguente:

```
D_Real apply(const std::vector<Geometry::Point2D>& pointVector,
 const D_Real _volume,
 const std::function<D_Real(Geometry::Point2D)>& Integrand) const;
```

dove _volume è il volume della cella e il vettore contiene i vertici che la delimitano.

Poiché questa parte esulava dallo scopo principale del progetto, le uniche regole di quadratura che sono state derivate da QuadratureRule sono quella

del punto medio, valida sia in 2D che in 3D che sulle fratture, e una regola a tre punti di ordine due valida però solo sui triangoli in due dimensioni. I loro nomi sono rispettivamente CentroidQuadrature e Triangle2D. La struttura del codice prevede però la possibilità di ampliare la scelta delle regole di quadratura con grande facilità.

È infine importante sottolineare che la classe Quadrature è una classe template che prende come parametri o Dimension<2> o Dimension<3>. Questa scelta ci permette di poter usare gli oggetti di questo tipo sia nel caso bidimensionale che nel caso tridimensionale. Anche la classe QuadratureRule gode della stessa generalità, infatti, il metodo del punto medio che da essa deriva, può essere chiamato sia in 2D che in 3D e anche esso ha bisogno di un parametro template. Il metodo Triangle2D, invece, deriva da QuadratureRule<Geometry::Dimension<2> > e quindi non ne ha bisogno.

## 3.7 Ulteriori osservazioni

Nel corso di questo progetto sono state utilizzate due librerie diverse: la libreria CGAL e la libreria Eigen. In mesh2D vengono utilizzati molti tipi geometrici provenienti dalle CGAL, che sono stati poi ereditati nella nostra struttura. Fra i più usati ci sono i Point_2<Kernel>, i Point_3<Kernel>, i Vector_2<Kernel>, i Vector_3<Kernel>, i Segment_2<Kernel> e i Segment_3<Kernel>. Queste classi implementano in maniera efficiente i concetti di punti, vettori e segmenti in due ed in tre dimensioni e le funzioni di base legate a questi concetti, quali la lunghezza di un vettore, il prodotto scalare, il prodotto scalare per vettore e la somma punto vettore. Per contro è obbligatorio usare -fpermessive per compilare usando queste librerie.
L'altra libreria che è stata utilizzata è la libreria Eigen che fornisce vettori e matrici molto efficienti e di cui abbiamo già parlato nel corso della relazione. Sia in un caso, che nell'altro, gli oggetti sono stati ridefiniti in un opportuno file di typedef e utilizzati con il nuovo nome. La stessa cosa è stata fatta per i numeri reali e per gli interi: Real e D_Real sono i nomi che sono stati dati ai numeri reali, nel primo caso i reali della libreria CGAL, nel secondo dei double, e UInt e D_UInt sono stati usati per denominare i long unsigned int. Avendo ridefinito in questo modo i numeri reali e gli interi è possibile cambiare solo a livello di typedef il tipo, e quest'ultimo viene cambiato in tutto il codice.

Come mostrato più volte in questa relazione, in questo progetto è stata ampiamente utilizzata la sintassi del nuovo standard C++11. Questa è stata usata non solo inserendo nel codice lambda functions o std::functions, ma inserendo anche tuple e la nuova sintassi per i cicli for: molto rapida

ed intuitiva. Quest'ultima permette infatti di sostituire la scrittura lunga e pedante di:

```
for (std::vector<Cell>::const_iterator cell_it =
    M_mesh.getCellsVector().begin();
        cell_it!=M_mesh.getCellsVector().end(); ++cell_it)
//do something
```

con la espressione molto più veloce e comprensibile

```
for (auto cell_it : M_mesh.getCellsVector())
//do something
```

Infine tutto il codice è stato commentato con un formato di tipo Doxygen, in modo tale da poter creare facilmente un file di tipo xml per capirne la struttura.

# Capitolo 4

# Test Numerici

Per verificare il corretto funzionamento del codice prodotto e la convergenza del metodo addottato, sono stati presi in considerazioni tre problemi test simili fra loro, di cui si conosce la soluzione esatta. Il problema che è stato risolto in tutti e tre i casi è il seguente:

$$\begin{cases} -\Delta u + u = f & \text{in } \Omega \\ u = h & \text{on } \Gamma_D \\ \dfrac{\partial u}{\partial \mathbf{n}} = g & \text{on } \Gamma_N \end{cases} \tag{4.1}$$

dove $\Omega = (0, \pi) \times (0, \pi)$, $\Gamma_D = 1 \cup 3$ e $\Gamma_N = 2 \cup 4$ con riferimento alla figura (4.1). Grazie al teorema di Lax-Milgram, è facile provare che questo teorema ammette un unica soluzione (si veda [4]). I casi che prenderemo in

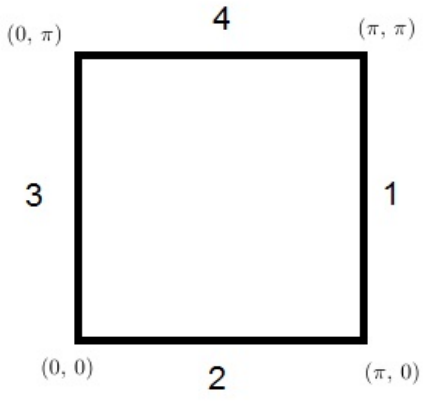



**Figura 4.1:** Dominio in cui sono stati risolti i casi test

esame sono i seguenti:

$$\begin{cases} f = 3\sin(x)\sin(y) & \text{in } \Omega \\ h = 0 & \text{on } \Gamma_D \\ g = -\sin(x) & \text{on } \Gamma_N \end{cases} \tag{4.2}$$

$$\begin{cases} f = 3\sin(x)\cos(y) & \text{in } \Omega \\ h = 0 & \text{on } \Gamma_D \\ g = 0 & \text{on } \Gamma_N \end{cases} \tag{4.3}$$

e

$$\begin{cases} f = 3\cos(x)\cos(y) & \text{in } \Omega \\ h = -\cos(y) & \text{on } 1 \\ h = \cos(y) & \text{on } 3 \\ g = 0 & \text{on } \Gamma_N \end{cases} \tag{4.4}$$

Sapendo che ognuno di questi problemi ammette un'unica soluzione è facile verificare che questa è

$$u = \sin(x)\sin(y)$$

nel primo caso,

$$u = \sin(x)\cos(y)$$

nel secondo e

$$u = \cos(x)\cos(y)$$

nel terzo. Abbiamo risolto con il nostro codice tutti e tre i casi e abbiamo ottenuto il grafico(4.2), che rappresenta la convergenza del metodo al tendere a zero del passo della griglia. L'andamento decrescente dell'errore in norma $L^2(\Omega)$ rappresentato sull'asse delle ordinate al tendere a zero del passo della griglia, raffigurato sull'asse delle ascisse, ci mostra che il metodo implementato funziona. In figura (4.4) si possono vedere i risultati ottenuti e la griglia utilizzata.

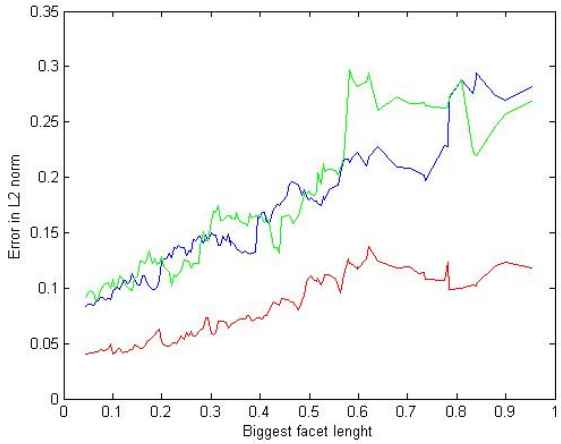



**Figura 4.2:** Grafco di convergenza per i tre casi test in esame

Per testare il funzionamento del codice anche nel caso di fratture si è risolto il seguente problema:

$$\begin{cases} -\nabla \cdot (K \nabla u) = 0 & \text{in } \Omega \\ \dfrac{\partial u}{\partial \mathbf{n}} = 0 & \text{on } 1,\ 4 \\ \dfrac{\partial u}{\partial \mathbf{n}} = 10 & \text{on } 3 \\ \dfrac{\partial u}{\partial \mathbf{n}} = -10 & \text{on } 6 \\ u = 10 & \text{on } 2 \\ u = 0 & \text{on } 5 \end{cases} \tag{4.5}$$

dove $\Omega = (0,\ 12) \times (0,\ 10)$ e i bordi del dominio sono come in figura (4.3).

Come si vede dalla figura (4.5), non solo le fratture influenzano la dinamica del sistema, ma riescono a essere visualizzate molto bene grazie al salvataggio in formato vtk e alla loro apertura in Paraview.

Il metodo utilizzato per risolvere il sistema lineare associato ai vari problemi test è chiamato SimplicialCholesky e si basa sulla fattorizzazione di Cholesky della matrice che, nei casi in esame, è sempre simmetrica e definita positiva. Questo metodo è built-in nella libreria Eigen ed è implementato per risolvere sistemi lineari con matrici sparse in maniera efficiente (si veda a tal proposito [5]).

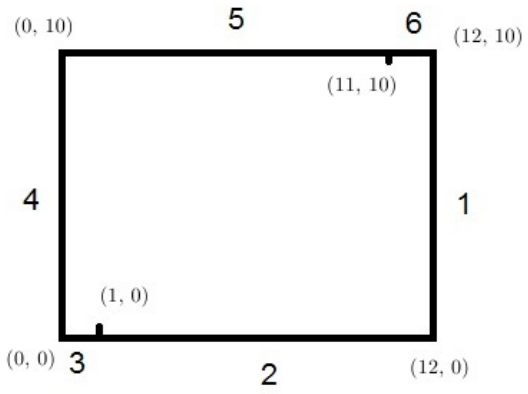



**Figura 4.3:** Dominio per il problema test con fratture.

Purtroppo, non essendo ancora disponibile al momento della stesura di questa relazione una versione della classe Mesh3D, il codice non è stato testato nel caso tridimensionale. Ci si augura di poterlo fare al più presto.

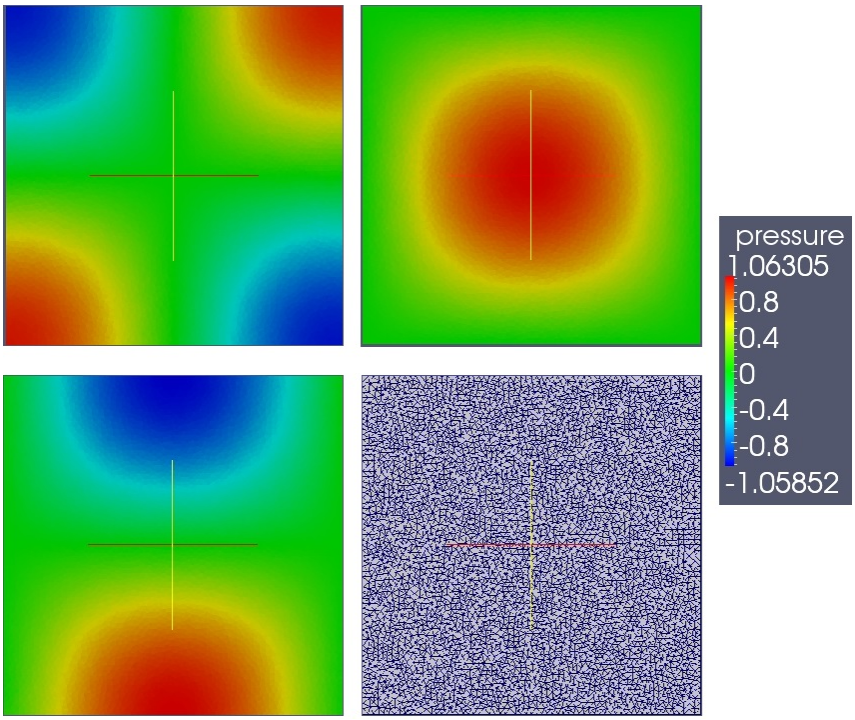



**Figura 4.4:** Grafico delle soluzioni ottenute: in alto a sinistra $u = \cos(x)\cos(y)$, in alto a destra $u = \sin(s)\sin(y)$, in basso a sinistra $u = \sin(x)\cos(y)$ e in basso a destra è raffigurata la griglia utilizzata.

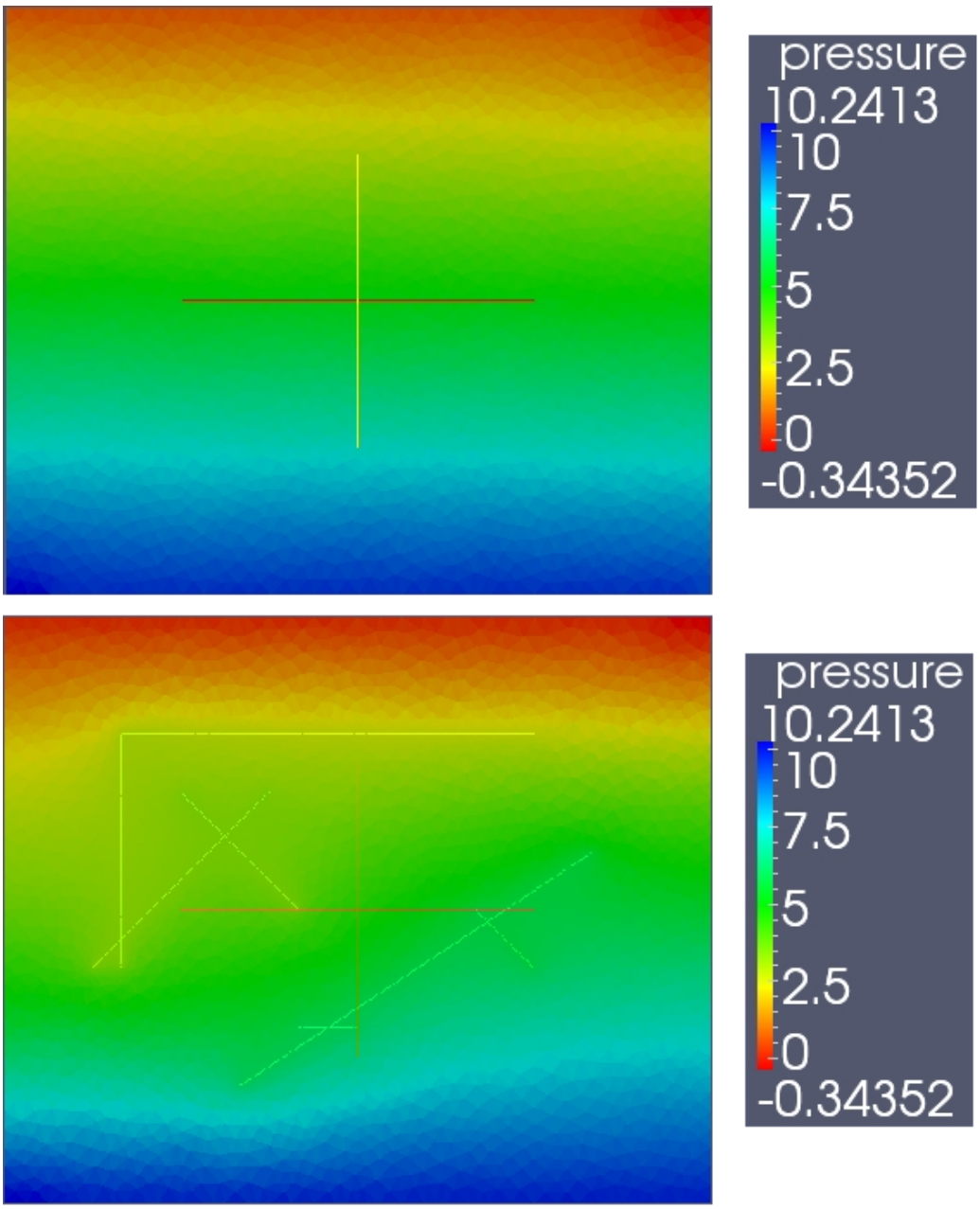



**Figura 4.5:** Confronto fra il problema risolto in presenza (a destra) ed in assenza (a sinistra) di fratture

# Capitolo 5

# Conclusioni e sviluppi futuri

In questo lavoro è stato sviluppato un codice che implementa alcuni importanti tool per la risoluzione del problema di Darcy in mezzi porosi con fratture. Innanzitutto è stata sviluppata una classe per contenere la griglia che è molto efficiente per l'assemblaggio delle matrici. Sono poi state sviluppate due classi che contengono le informazioni necessarie per passare le condizioni al contorno ad una terza classe programmata da noi, che assembla la matrice di stiffness con una discretizzazione di tipo volumi finiti descritta in [1]. Infine è stato sviluppato un oggetto in grado di riempire la matrice di massa per la discretizzazione a volumi finiti e una serie di tool per calcolare gli integrali delle forzanti e della soluzione sulla griglia data. Il codice sviluppato ha dimostrato di convergere in alcuni casi test.

Tutto il lavoro svolto, permette di essere applicato a problemi anche di natura molto diversa rispetto al problema di Darcy. Inoltre, si è cercato di rendere il codice compatibile con discretizzazioni diverse rispetto a quella a volumi finiti. A tal proposito, fra le altre cose, è stata creata la classe MatrixHandler, dalla quale ereditare un possibile assemblatore per matrice di rigidezza diverso da quello a volumi finiti.

Il lavoro fatto può essere ampliato in molti modi diversi: si potrebbe innanzitutto generalizzare il problema da risolvere al caso in cui il fluido considerato sia semicomprimibile. Un ulteriore sviluppo consiste nel generalizzare ulteriormente il fenomeno in considerazione esaminando un fluido bifase e considerando una permeabilità anisotropa. Si potrebbe infine ampliare la scelta di formule di quadratura derivandole dalla classe QuadratureRule.

# Bibliografia


[1] M.Karimi-Fard, L.J. Durlofsky, K. Aziz, *An Efficient Discrete-Fracture Model Applicable for General-Purpose Reservoir Simulators*. SPE Journal, Volume9, Number2, 2004.

[2] S.B. Lippmani, J. Lajoie, B.E. Moo, *C++Primer, IV edeizione*. Addison Wesley, 2005.

[3] A. Quarteroni,*Modellistica numerica per problemi differenziali, IV edeizione*. Springer, 2008.

[4] S. Salsa,*Equazioni a derivate parziali. Metodi, modelli e applicazioni, II edeizione*. Springer, 2010.

[5] *Eigen 3.1 manual, www.eigen.tuxfamily.org.*

[6] *CGAL 4.2 manual, www.cgal.org.*