Introduzione

Il presente progetto si inserisce nell'ambito dello studio del moto di un fluido in un mezzo poroso.[1] Lo scopo è quello di studiare e risolvere il problema nel caso di un network di fratture che si intersecano.
Il nostro progetto parte da un codice già esistente in C++ in grado di risolvere il problema di Darcy nel caso in cui due fratture abbiano un'intersezione a *X*. [4] Il nostro obiettivo è quello di estendere il codice ad altri tipi di intersezione, in particolare vogliamo risolvere il problema in presenza di tre fratture che si intersechino in un unico pinto comune formando una *Y*.
Per l'implementazione e la risoluzione del problema tramite elementi finiti, abbiamo utilizzato principalmente la libreria `GetFEM++`, una serie di liberie scritte in C++ che mettono a disposizione dell'utente una serie di funzioni utilizzabili per la soluzione dei problemi FEM.
Il modello fisico su cui si basa la nostra implementazione è un modello ridotto, considera cioè le fratture come delle grandezze 1d, trascurandone lo spessore. Per poter ricavare le condizioni d'interfaccia nel caso della nuova intersezione si è reso necessario ritornare al modello 2d, e quindi considerare l'intersezione come un triangolo e non come un punto. [3] [2]

Indice

1 Mezzi porosi e legge di Darcy **3**
1.1 Legge di Darcy 3
1.2 Formulazione debole del problema 4

2 Considerazioni preliminari **6**
2.1 Strumenti di sviluppo 6
2.1.1 `Git` 6
2.1.2 `CMake` 6
2.1.3 `Doxygen` 9
2.2 Dati del problema 10
2.2.1 Intersezioni 10
2.2.2 File data 11

3 Implementazione delle fratture **14**
3.1 La Classe `FracturesSet` 15
3.2 La Classe `FractureHandler` 16
3.3 Le Classi `LevelSetHandler` e `LevelSetData` 18
3.4 Le Classi `FractureIntersect` e `IntersectData` 19
3.5 Le Classi `BC` e `BCHandler` 21

4 Classi per la gestione dell'intersezione **22**
4.1 La Classe `MatrixBifurcationHandler` 24
4.2 La Classe `Intersection` 25
4.3 Classe `PointData` e Classe `TriangleData` 26

5 Soluzione del problema numerico **27**
5.1 La Classe `DarcyFracture` 28

6 Risultati numerici e validazione del modello **30**
6.1 Biforcazione semplice, 3 fratture 30
6.2 Doppia biforcazione e cross, 6 fratture 31

6.3 Doppia biforcazione verticale, 5 fratture 31
6.4 Caso limite 31
6.5 Conclusioni 31

Capitolo 1

Mezzi porosi e legge di Darcy

I materiali porosi sono dei mezzi in cui si possono distinguere due costituenti: una matrice solida, la parte che costituisce la struttura rigida del corpo, e lo spazio vuoto restante, che può essere riempito con uno o più fluidi.
Risulta spesso interessante studiare il moto di un fluido in un mezzo poroso in cui sia possibile distinguere delle fratture o dei canali al suo interno. Nell'analisi del moto di un fluido in una frattura tra diversi strati geologici, ad esempio, si può pensare che il fluido, oltre a insinuarsi e scorrere nella frattura, si propaghi mediante filtrazione negli strati adiacenti la frattura stessa. Un'altra possibile applicazione in ambito idrogeologico riguarda lo studio di come i fiumi irrighino il terreno ad essi circostante.

Il moto del fluido all'interno del mezzo poroso viene descritto tramite la legge di Darcy. Generalmente le dimensioni delle fratture sono di molto inferiori a quelle del dominio occupato dal mezzo poroso. Questo ha portato allo sviluppo di tecniche per lo studio del flusso all'interno delle fratture basate su modelli ridotti. Questi modelli si inseriscono nella categoria dei modelli multiscala, ed hanno il vantaggio di evitare la risoluzione del campo di moto sulle scale spaziali molto piccole nella frattura.

1.1 Legge di Darcy

La legge di Darcy descrive la filtrazione di un fluido incomprimibile all'interno di un mezzo poroso. In particolare costituisce un legame tra il campo di velocità del fluido e il gradiente di pressione nel mezzo poroso.
Il moto di un fluido incomprimibile in un mezzo poroso (considerando z=0 come quota di riferimento) può essere descritto dalla seguente coppia di equazioni:

$$\mathbf{u} = -\frac{\mathbf{K}}{\mu}\nabla p^1 \tag{1.1}$$

$$\nabla \cdot u = f \tag{1.2}$$

dove:

- $\mathbf{u}(\mathbf{x}, \mathrm{t})$ è il campo di velocità macroscopica del fluido misurata in $[m/s]$;
- $\mathbf{K}(\mathbf{x})$ è il tensore simmetrico di permeabilità assoluta misurato in $[m^2]$;
- $p(x,t)$ è la pressione del fluido in $[Pa]$=$[N/m^2]$;
- μ $(\mathbf{x}, \mathrm{t})$ rappresenta la viscosità dinamica del fluido in $[Pa\ s]$, che per semplicità verrà considerata costante;
- $f(\mathbf{x}, \mathrm{t})$ rappresenta il termine sorgente di dimensione $[s^{-1}]$.

La prima equazione è la legge di Darcy e deriva dalla conservazione del momento nelle equazioni di Navier-Stokes con opportuni passaggi, mentre la seconda rappresenta la conservazione della massa.
L'approssimazione con il modello di Darcy è valida per fluidi Newtoniani con bassi valori del numero di *Raynolds* ($Re < 10$), per cui gli effetti inerziali possono essere trascurati. Per valori maggiori è necessario estendere il modello poichè l'equazione 1.1 presenta dei limiti.

1.2 Formulazione debole del problema

Il problema del flusso di un fluido in un mezzo poroso assume la seguente forma:

$$\begin{cases} \mathbf{K}^{-1}\mathbf{u} + \nabla p = 0 \\ \nabla \cdot u = f \end{cases} \tag{1.3}$$

con condizioni al contorno:

$$\begin{cases} \mathbf{u} \cdot \mathbf{n}_\Omega = 0 & su\ \Gamma^u \\ p = g & su\ \Gamma^p \end{cases} \tag{1.4}$$

Per ricavare la formulazione debole del problema, così da poter discretizzare con gli elementi finiti, moltiplichiamo la prima equazione del sistema 1.2 per una funzione test $\mathbf{v}$, la seconda per una funzione test q e integriamo

[1] Più avanti indicheremo $\frac{\mathbf{K}}{\mu}$ con $\mathbf{K}$ per semplicità

su Ω.
La formulazione debole diventa: trovare ($\mathbf{u}$, p) tali che:

$$\begin{aligned} \int_\Omega (\mathbf{K}^{-1}\mathbf{u}) \cdot \mathbf{v}\, dx - \int_\Omega p \nabla \cdot \mathbf{v}\, dx + \int_{\Gamma^p} g\, \mathbf{u} \cdot \mathbf{n}\, d\Gamma = 0 \quad \forall\, \mathbf{v} \in \mathbf{W} \\ \int_\Omega \nabla \cdot \mathbf{u} q\, dx = \int_\Omega f q\, dx \quad \forall\, q \in Q \end{aligned} \tag{1.5}$$

dove:

$$\begin{aligned} \mathbf{W} &= H_{div}(\Omega) = \left\{ \mathbf{w} \in [L^2(\Omega)]^2, \nabla \cdot \mathbf{w} \in L^2(\Omega), \mathbf{v} \cdot \mathbf{n} = 0\, su\, \Gamma^n \right\} \\ Q &= L^2(\Omega) \end{aligned}$$

Per la discretizzazione in spazio scegliamo lo spazio degli elementi finiti di grado due per la velocità e uno per la pressione e ricaviamo la seguente formulazione algebrica:

$$\begin{cases} A11\mathbf{U} + A12\mathbf{P} = F_1 \\ A12^T\mathbf{U} = F_2 \end{cases} \tag{1.6}$$

dove $A11 \in \mathbb{R}^{N_u \times N_u}$ e $A12 \in \mathbb{R}^{N_p}$ sono le matrici realative alle forme bilineari $a(\cdot,\cdot)$ e $b(\cdot,\cdot)$ definite come:

$$a(\mathbf{u}_h, \mathbf{v}_h) = \int_\Omega (\mathbf{K}^{-1}\mathbf{u}_h) \cdot \mathbf{v}_h\, dx \qquad b(\mathbf{u}_h, q) = -\int_\Omega q_h \nabla \cdot \mathbf{u}_h\, dx$$

$$f1(\mathbf{u}_h) = \int_{\Gamma^p} g_h\, \mathbf{u}_h \cdot \mathbf{n}\, d\Gamma \qquad f2(q_h) = -\int_\Omega f q\, dx$$

Nel nostro caso quindi, per risolvere il flusso di un fluido all'interno di una frattura in un mezzo poroso si ottiene il seguente sistema algebrico:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{12}^T & 0 \end{bmatrix} \begin{bmatrix} U \\ P \end{bmatrix} = \begin{bmatrix} F_1 \\ F_2 \end{bmatrix} \tag{1.7}$$

Se vi sono più fratture che si intersecano è necessario imporre delle condizioni che leghino i rispettivi flussi e le rispettive pressioni. Il codice da cui siamo partite impone le condizioni d'interfaccia nel caso di un'intersezione di tipo *Cross* in modo debole. Il nostro codice, come verrà spiegato più avanti, impone le condizioni d'interfaccia nel caso della *Biforcazione* in maniera forte, ossia una volta che il sistema algebrico è già stato assemblato.

Capitolo 2

Considerazioni preliminari

2.1 Strumenti di sviluppo

Lo sviluppo del nostro progetto ha richiesto l'uso di strumenti specifici atti a facilitare e sistematizzare la gestione del codice quali `git`, `CMake` e `Dxygen`.

2.1.1 `Git`

`Git` è un sistema di controllo versione utilizzabile direttamente da linea di comando, molto diffuso è utile per tenere traccia delle varie fasi di sviluppo del codice. Git gestisce in modo adeguato i contributi al codice provenienti da agenti esterni e permette la condivisione del codice.
Il codice del progetto è reperibile su `git` ed è possibile scaricarlo e collabolare allo sviluppo clonando il codice dalla repository `GitHub`:

```
git clone https://github.com/mariaiemoli/progetto.git
```

Nella cartella principale è contenuto anche un file `.gitignore`, in cui sono specificate le estensioni dei files e le sottocartelle che non devono essere visionati in una repository `git`. In particolare non si è interessati ai files temporanei che vengono eventualmente generati dagli editor, e alle cartelle generate da uno scorretta configurazione di `Doxygen`.

2.1.2 `CMake`

`CMake` è un software libero multipiattaforma nato per l'automazione dello sviluppo. Sostanzialmente con l'uso di `CMake` è possibile configurare e generare Makefile per il sistema operativo in uso. Questo facilita la diffusione del codice tra sviluppatori, non dovendo far altro che lanciare `CMake` e lasciare che sia lui ad occuparsi della ricerca di compilatori e librerie locali e della

costruzione di software.
`CMake` si basa sulla creazione di files `CMakeLists.txt` nella directory del progetto, che contengono le direttive necessarie per creare il `Makefile` per compilare ed eseguire il codice.
Nella directory principale si trova un primo file `CMakeLists.txt` in cui si impostano i parametri fondamentali quali la dipendenza dalla versione di `CMake`, il nome del progetto, si includono le directory dei sorgenti nel path di compilazione e si caricano le librerie esterne utilizzate dai sorgenti.
le librerie che vengono caricate sono:

- `Eigen`, libreria template per l'algebra lineare usata principalmente nella fase di imposizione delle condizioni di interfaccia sul triangolo di intersezione nel caso della biforcazione, per il modello ridotto;
- `Getfem++`, libreria matematica per gli elementi finiti usata per scrivere e risolvere il problema numerico;
- `Blas` (Basic Linear Algebra Subprograms) e `Qhull`, librerie necessarie per `Getfem++`;
- `Doxygen`, usato per generare la documentazione

```
CMAKE_MINIMUM_REQUIRED( VERSION 2.8 )

PROJECT(PACS)

SET(CMAKE_CXX_FLAGS "-std=c++0x -Wall ${CMAKE_CXX_FLAGS}")

SET(CMAKE_MODULE_PATH
  ${CMAKE_SOURCE_DIR}/cmake ${CMAKE_MODULE_PATH})

# Include Eigen3
FIND_PACKAGE(PkgConfig)
PKG_CHECK_MODULES(EIGEN3 REQUIRED eigen3)
INCLUDE_DIRECTORIES(${EIGEN3_INCLUDE_DIRS})

# Include Doxygen
find_package(Doxygen)
[ ... ]

# Include Getfem
if (GETFEM_LIBRARIES AND GETFEM_INCLUDE_DIRS)
[ ... ]

#Include for BLAS library
find_package ( BLAS )
[ ... ]
```

```
#Include for LAPACK library
find_package ( LAPACK )
[ ... ]

#Include for QHULL library
set(QHULL_MAJOR_VERSION 6)
[ ... ]

SET(CMAKE_INSTALL_PREFIX
${CMAKE_SOURCE_DIR}/install-dir CACHE PATH "" FORCE)

ADD_SUBDIRECTORY(src)
ADD_SUBDIRECTORY(test)
```

Per poter usare `CMake` è necessario posizionarsi nella directory in cui è contenuto il progetto e creare una cartella in cui compilare il codice. Per lanciare `CMake` è necessario digitare i seguenti comandi:

```
cd <build_dir>
cmake <source_dir>
```

`CMake` imposta le variabili di path come definito nel file `CMakeLists.txt`, esamina le dipendenze da librerie esterne e procede esaminando le sottocartelle aggiunte nel file di configurazione. In particolare esamina le sottocartelle `src` e `test`, in cui sono presenti altri files `CMakeLists.txt`, che impostano i rispettivi obiettivi e definiscono le rispettive sottocartelle. Questo permette a `CMake` di esaminare tutta la directory in cui è contenuto il progetto in maniera ricorsiva.

Nella cartella test si trovano delle sottocartelle ognuna delle quali contenente un file data di esempio per poter eseguire il codice e un file `CMakeLists.txt` . Il file `CMakeLists.txt` nella cartella dei test definisce gli obiettivi e i collegamenti necessari per l'esecuzione.

Una volta che il `Makefile` è stato generato è possibile compilare il codice con il comando:

```
make
```

Questo crea in ogni sottocartella della cartella `test` un eseguibile chiamato `Test`. Per poterlo eseguire è sufficiente lanciare da terminale, dopo essersi posizionati nella directory corretta,

```
.\Test-iesimo
```

2.1.3 `Doxygen`

`Doxygen` è un sistema multipiattaforma molto diffuso per la generazione automatica della documentazione di un codice. Per ottenere la documentazione è necessario introdurre nel codice dei commenti con una sintassi particolare, e il risultato contiene l'elenco delle classi implementate, i loro diagrammi di collaborazione e la descrizione di metodi ed attributi. Per poter creare la documentazione al codice con `Doxygen` è necessario che nella cartella principale sia presente il file `Doxygen`, in caso contrario è necessario crearlo con il comando:

```
Doxygen -g
```

In questo file sono definite le variabili che permettono di creare la documentazione come il nome del progetto, la directory dove creare la documentazione, le cartelle dove si trovano i sorgenti e l'estensione dei files da considerare.

```
PROJECT_NAME      = Problema di Darcy in un network di fratture
OUTPUT_DIRECTORY  = ./build/doc
INPUT             = ./src ./test
FILE_PATTERNS     = *.cc *.h
HAVE_DOT          = YES
COLLABORATION_GRAPH = YES
HIDE_UNDOC_RELATIONS = NO
```

Dopo aver lanciato `CMake` è possibile generare la documentazione semplicemente eseguendo:

```
cd <build_dir>
make doc
```

Questo crea una sottocartella `build/doc` che contiene la documentazione `html` e LaTeX.

2.2 Dati del problema

2.2.1 Intersezioni

Come è stato precedentemente detto, il codice di partenza trattava un tipo particolare di intersezione tra fratture, da noi chiamato *Cross*. Questo tipo di intersezione ha la seguente forma:

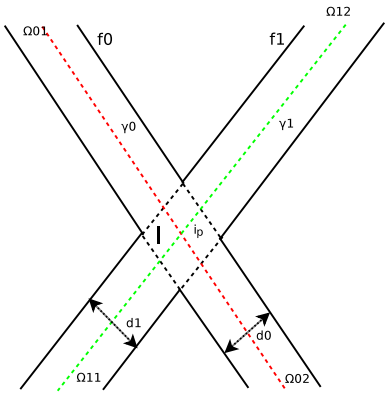

Figura 2.1: Struttura di un'intersezione di tipo *Cross* tra due fratture.

L'area di intersezione è constituita da un parallelogramma che suddivide ogni frattura in due zone Ω_{i1} e Ω_{i2}. Il modello su cui noi ci basiamo riduce la frattura, qui rappresentata in un dominio $\mathbb{R}^2$, alla retta di sostegno γ_i. In questo modo l'intersezione si riduce ad un punto.
Il tipo di intersezione su cui ci siamo concentrate noi, invece, ha la seguente forma:

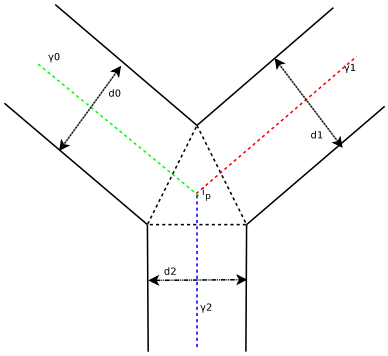

Figura 2.2: Struttura di un'intersezione di tipo *Bifurcation* tra due fratture.

In questo caso le fratture coinvolte sono tre e la zona interessata dall'intersezione è rappresentata da un triangolo. Anche qui consideriamo un modello

ridotto e le fratture sono individuate dalle rette di sostegno γ_i in $\mathbb{R}$. Le condizioni d'interfaccia che vengono ricavate in questo caso valgono solo nel caso in cui le tre fratture si intersechino in un unico punto comune i_p e tale punto sia contenuto nel triangolo d'intersezione nella rappresentazione 2d.

2.2.2 File data

I dati per la definizione del problema sono definiti nei file .txt contenuti nella sottocartella `dati`.
Ognuno di questi files ha la seguente struttura:

- una prima parte dove vengono definite la directory dove trovare l'eventuale mesh da importare, la directory dove salvare i risultati e il numero di fratture che entrano in gioco;
- una sezione dedicata a parametri e grandezze necessari per la definizione del dominio del mezzo;
- una sezione per ogni frattura con la dichiarazione del level set, dei parametri che individuano il segmento che rappresenta la frattura e delle grandezze necessarie per risolvere il problema di Darcy.

Listing 2.1: Definizione del dominio

```
meshFile = meshes/mesh  // directory da cui importare eventualmente la mesh
folderVTK = ./vtk/    // directory dove salvare i risultati

numberFractures = 3

[mediumData]     // informazioni legate al mezzo poroso in questione

   [./domain]     // informazioni necessarie per costruire o importare la mesh
      [ ... ]
   [../]

   [./darcy]     // informazioni sulla natura permeabile del mezzo
      invK = 1.
      invKDist11 = 1.
      invKDist12 = 0.
      invKDist22 = 1.
      [...]
   [../]

[../]
```

Nonostante il nostro problema si concentri solamente sulla soluzione del flusso all'interno delle fratture, noi costruiamo comunque la mesh relativa al mezzo. Questa mesh verrà usata solamente come supporto, sarà puramente

accessoria nel corso dell'implementazione.
Per quanto riguarda la permeabilità, poichè le fratture sono costituite dallo stesso materiale, questa resta uguale in tutto il dominio, sia nel mezzo che nelle fratture.

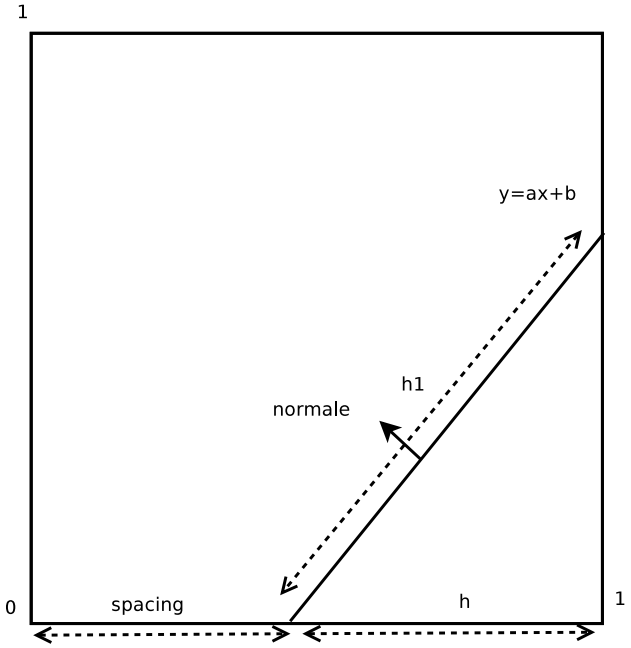

Figura 2.3: Principali parametri di definizione per una frattura

Listing 2.2: Definizione della frattura

```
[fractureData]

spaceDimension = 1.

    [./levelSet]  // parametri di definizione del level set
      // funzione level set che rappresenta la frattura
      levelSet = y-a*x-b
      ylevelSet = y-a1*t-b1
      xlevelSet = x-a2*t-b2
      levelSetCut = -1
      yMap = a1*t+b1
      xMap = a2*t+b2
      // fattore di scala, rapporto tra la lunghezza della mesh di integrazione e la
      // lunghezza della mesh reale
      jacMap = [ h/h1 ]
      normalMap = [ ]
      integrationTypeSimplex = IM_STRUCTURED_COMPOSITE(IM_TRIANGLE(3),1)
    [../]
```

```
  [./domain]  // parametri di definizione del dominio di integrazione
    thickness =
    spacing = x+c
    spatialDiscretization = n
    // dimensioni del dominio
    translateAbscissa = c
    lengthAbscissa = h1
    lengthOrdinate = 0.
    lengthQuota = 0.
    meshType = GT_PK(1,1)
    // variabili per l'integrazione
    integrationTypeVelocity = IM_GAUSS1D(3)
    integrationTypePressure = IM_GAUSS1D(2)
    FEMTypeVelocity = FEM_PK(1,1)
    FEMTypePressure = FEM_PK(1,0)
    FEMTypeLinear = FEM_PK(1,1)
  [../]

  [./darcy]  // parametri per la definizione del problema di Darcy
    etaNormal = 0.01
    etaNormalDistribution = 1.
    etaTangential = 100.
    etaTangentialDistribution = 1.
    source =
    solution =
    velocity =
  [../]

[../]
```

Capitolo 3

Implementazione delle fratture

Passiamo ora a descrivere le principali classi del codice che definiscono la struttura e le proprietà delle fratture, porgendo particolare attenzione alle classi modificate e a quelle da noi implementate.

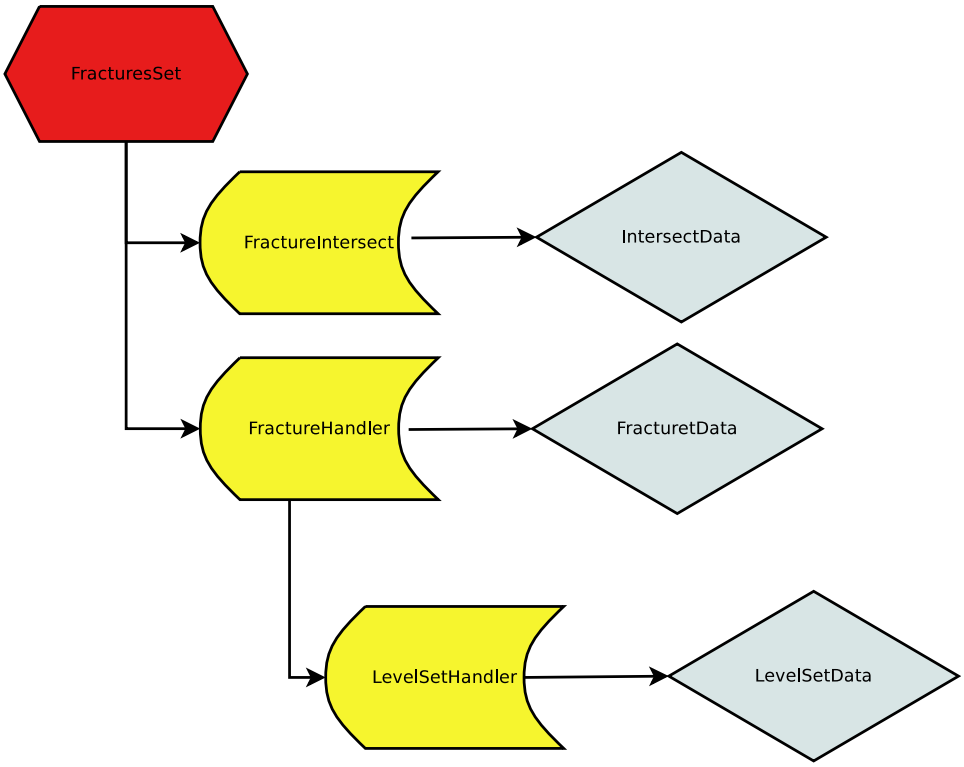

Figura 3.1: Inclusione tra le varie classi che costituiscono l'insieme delle fratture

Le classi usate per rappresentare le fratture e le intersezioni si dividono in due categorie:

- le classi in cui sono implementati tutti i metodi necessari per la costruzione e la manipolazione della grandezza (insieme delle fratture, singola frattura, intersezioni, level set). Queste classi sono principalmente denominate col il suffisso *Handler* e costituiscono l'ossatura dell'insieme delle fratture;

- le classi in cui sono contenuti tutti i parametri geometrici, fisici e le grandezze numeriche per l'implementazione e la risoluzione con elementi finiti, che riguardano la grandezza in esame. Vengono indicate con il suffisso *Data* e rappresentano sempre un campo della corrispondente classe di tipo *Handler*.

3.1 La Classe `FracturesSet`

La class `FracturesSet` è la classe che contiene tutte le fratture e le rispettive intersezioni. La funzione che costruisce l'insieme delle fratture e le eventuali intersezioni è la funzione `init`, che prende in input tutti i parametri necessari quali il numero di fratture, le mesh per l'integrazione e una variabile di tipo GetPot per la lettura dal file data.

Listing 3.1: Classe `FracturesSet`

```
class FracturesSet
{
public:

    enum
    {
        FRACTURE_UNCUT = 10000,
        FRACTURE_INTERSECT = 10000
    };

    FractureHandler ( const GetPot& dataFile,
                      const size_type& ID,
                      const std::string& section = "fractureData/" );

    void init ( );

    [ ... ]

private:
        FracturePtrContainer_Type M_fractures;

        FractureIntersectPtr_Type M_intersections;
        [ .... ]
```

```
};
```

I campi principali della classe sono:

- `M_fractures`, variabile di tipo vettore di puntatori alla classe `FractureHandler` di cui parleremo più avanti, che rappresenta l'insieme di tutte le fratture;

- `M_intersections`, un puntatore alla classe `FractureIntersect`, che rappresenta l'insieme di tutte le intersezioni.

3.2 La Classe `FractureHandler`

La classe `FractureHandler` è la classe che inizializza e gestisce ogni frattura. I principali campi di questa classe sono:

- `M_data`: classe `FractureData` che contiene tutte le informazioni circa la natura geometrica e fisica della singola frattura;

- `M_levelSet`: puntatore alla classe `LevelSetHandler` di cui parleremo più avanti;

- i metodi di integrazione e le mesh per pressione e velocità;

- il vettore dei gradi di libertà estesi per velocità e pressione.

La particolarità di questa classe è che possiede due mesh: una mesh 2d `M_meshMapped` e una mesh 1d `M_meshFlat`. Questo deriva dal fatto che le fratture sono su un piano e hanno una rappresentazione del tipo $y = f(x)$, cioè i loro punti hanno coordinate (x, y). La libreria che noi usiamo, *GetFEM++*, non è in grado di integrare su una mesh i cui punti hanno coordinate (x, y) un'equazione 1d come quella del nostro modello ridotto. La tecnica è quindi quella di risolvere il problema integrando sulle mesh piatte 1d ottenute proiettando le mesh reali, e successivamente interpolare i risultati ottenuti per ritornare sulle mesh 2d. La creazione delle mesh è nella funzione `init`.

La funzione principale di questa classe è `setMeshLevelSetFracture` . Tale funzione crea un legame tra la frattura corrente e le fratture con cui ha un'intersezione, tenendo traccia della mesh e dei valori del level set della frattura corrente nei punti della frattura intersecata. Nel caso di intersezione di tipo *Cross*, dove l'intersezione tra due level set non è detto che avvenga su due punti delle rispettive mesh, vengono aggiunti i gradi di libertà estesi, due per la velocità e uno per la pressione. Nel caso dell'intersezione di tipo *Bifurcation* l'introduzione di tali elementi non si rende necessaria.

Listing 3.2: Funzione `setMeshLevelSetFracture`

```
size_type FractureHandler::setMeshLevelSetFracture (
                                                FractureHandler& otherFracture,
                                                size_type& globalIndex,
                                                 const std::string& type )
{
[ ... ]
if ( !M_meshLevelSetIntersect[ otherFractureId ].get() )
{
    M_meshLevelSetIntersect[ otherFractureId ].reset
          ( new GFMeshLevelSet_Type ( M_meshFlat ) );
    LevelSetHandlerPtr_Type otherLevelSet = otherFracture.getLevelSet();
    M_levelSetIntersect [ otherFractureId ].reset
          ( new GFLevelSet_Type ( M_meshFlat, 1, false ) );
    M_levelSetIntersect [ otherFractureId ]->reinit();

    const size_type nbDof =
          M_levelSetIntersect [ otherFractureId ]->get_mesh_fem().nb_basic_dof();

    for ( size_type d = 0; d < nbDof; ++d )
    {
        base_node node = M_levelSetIntersect [ otherFractureId ]
                          ->get_mesh_fem().point_of_basic_dof(d);
        base_node mappedNode ( node.size() +1 );
        scalar_type t = d*1./(M_data.getSpatialDiscretization () );
        base_node P (node.size());
        P [0] = t;
        mappedNode [0] = node [0];
        mappedNode [1] = M_levelSet->getData()->y_map ( P );
        M_levelSetIntersect [ otherFractureId ]->values(0)[d] =
          otherLevelSet->getData()->ylevelSetFunction ( mappedNode );
    }

    M_meshLevelSetIntersect[ otherFractureId ]
          ->add_level_set ( *M_levelSetIntersect [ otherFractureId ] );
    M_meshLevelSetIntersect[ otherFractureId ]->adapt ();

    size_type i_cv = 0;
    dal::bit_vector bc_cv =
    M_meshLevelSetIntersect[ otherFractureId ]->linked_mesh().convex_index();

    for ( i_cv << bc_cv; i_cv != size_type(-1); i_cv << bc_cv )
    {
      if ( M_meshLevelSetIntersect[ otherFractureId ]->is_convex_cut ( i_cv ) )
      {
        if ( type == "Cross")
        {
            M_meshFlat.region ( FractureHandler::FRACTURE_UNCUT
                + ( M_ID + 1 ) ).sup ( i_cv );
```

```
            M_meshFlat.region ( FractureHandler::FRACTURE_INTERSECT
                    * ( M_ID + 1 ) + otherFractureId + 1 ).add( i_cv );
            M_extendedPressure.push_back (
                    M_meshFEMPressure.ind_basic_dof_of_element ( i_cv )[0] );
            M_extendedVelocity.push_back (
                    M_meshFEMVelocity.ind_basic_dof_of_element ( i_cv )[0] );
            M_extendedVelocity.push_back (
                    M_meshFEMVelocity.ind_basic_dof_of_element ( i_cv )[1] );
          }

        M_fractureIntersectElements [ otherFractureId ].push_back ( i_cv );

            [ ... ]
      }
    }
}

return numIntersect;
}
```

3.3 Le Classi `LevelSetHandler` e `LevelSetData`

Le fratture sono rappresentate come delle rette, funzioni $y = f(x) = ax + c$. Ad ogni frattura è associata una funzione level set del tipo $f(x, y) = y - ax - b$ che divide il piano in due semipiani: i punti un cui $f(x, y) > 0$ e i punti in cui $f(x, y) < 0$. I punti in cui $f(x, y) = 0$ sono quelli in cui è definita la frattura.

La classe `LevelSetHandler` inizializza il levelset associato alla frattura. La funzione fondamentale della classe è la funzione `init`, funzione che inizializza il level set, definisce i metodi di integrazione e aggiunge alla mesh di supporto l'informazione legata al level set. I campi fondamentali della classe sono:

- `M_data`: puntatore alla classe `LevelSetData`, classe che contiene tutte le informazioni sul level set e le funzioni per valutarne il valore nei punti;

- `M_mesh`: puntatore ad una variabile di tipo *getfem::mesh_level_set*, classe che contiene informazioni sulla mesh tagliata da level set;

- `M_levelSet`: oggetto di tipo *getfem::level_set*, variabile che definisce il level set. In GetFEM un level set è rappresentato come una o due funzioni scalari, definito su una mesh di tipo *getfem::mesh_fem*, ossia una mesh su cui è definito un metodo ad elementi finiti.

Listing 3.3: Classe `LevelSetHandler`

```
class LevelSetHandler
{
public:

    LevelSetHandler ( const GetPot& dataFile,
                      const std::string& section = "fractureData/",
                      const std::string& sectionLevelSet = "levelSet/" );

    void init ( getfem::mesh& mediumMesh,
                const std::string& mediumIntegrationTypeVelocity,
                const getfem::mesh_fem& mediumMeshFEMPressure,
                const getfem::mesh_fem& mediumMeshFEMVelocity );

        [ ... ]
private:

    LevelSetDataPtr_Type M_data;

    GFMeshLevelSetPtr_Type M_mesh;

    GFLevelSetPtr_Type M_levelSet;
        [ ... ]
};
```

3.4 Le Classi `FractureIntersect` e `IntersectData`

La classe `FractureIntersect` rappresenta tutte le intersezioni tra le fratture. Ad ogni tipo di intersezione viene associato il vettore delle classi `IntersectData`, classe che rappresenta la vera e propria intersezione, tenendo traccia delle informazioni sulle fratture e dell'id dell'elemento della mesh di supporto in cui avviene l'incontro.

Le intersezioni vengono classificate come *Parallel*, *Cross* e *Bifurcation*. L'idea alla base della classificazione è quella di considerare gli elementi della mesh 2d di supporto, la mesh del mezzo, in cui passano due o più fratture, e di associargli un flag:

- *Parallel*: quando due o più fratture passano in uno stesso elemento della mesh ma non si intersecano;
- *Cross*: quando due fratture passano nello stesso elemento della mesh e si intersecano formando una X;
- *Bifurcation*: quando tre fratture si intersecano in un punto comune formando una Y.

Listing 3.4: Funzione `constructIntesection`

```
void FractureIntersect::constructIntesection (
        const getfem::mesh& mesh, getfem::mesh_level_set& meshLevelSet,
        const FracturePtrContainer_Type& fractures )
{
[ ... ]
meshLevelSet.find_level_set_potential_intersections (
                            listOfConvex, listOfLevelSet_bitVector );

sizeVectorContainer_Type listOfLevelSet ( listOfConvex.size() );

if( listOfConvex.size() > 0 )
{
   for ( size_type i = 0 ; i < listOfConvex.size(); ++i )
   {
      // Conversione da un vettore di bit a un vettore di interi
      fromBitVectorToStdVector (
      listOfLevelSet_bitVector [ i ], listOfLevelSet [ i ] );

       // Per ogni elemento della mesh di supporto in cui passano
      // almeno due fratture verifico il tipo di intersezione
       IntersectionType type = intersectionType (
                meshLevelSet, listOfConvex [ i ], listOfLevelSet [ i ] );

       // Prendo i puntatori alle fratture coinvolte
      FracturePtrContainer_Type fracturesInvolved ( listOfLevelSet[i].size() );

      for ( size_type f = 0; f < fracturesInvolved.size(); ++f )
      {
         fracturesInvolved [ f ] = fractures [ listOfLevelSet [ i ] [ f ] ];
       }
       // Costruisco la classe IntersectData per la nuova intersezione
       // e la aggiungo in base al tipo
      IntersectData intersection;
      intersection.setIntersection ( listOfConvex[i], fracturesInvolved );
      M_intersections [ type ].push_back ( intersection );
    }
[ ... ]
}
```

La funzione `constructIntesection` contiene la classificazione delle intersezioni. In questa funzione viene data una numerazione globale alle intersezioni, prima quelle di tipo *Cross* e poi quelle di tipo *Bifurcation*. Nel caso delle prime vengono aggiunte due nuove incognite, una per ogni frattura, necessarie per imporre le condizioni d'interfaccia nella formulazione debole del problema. Queste nuove variabili rappresentano la pressione nel punto di intersezione e vengono poi accoppiate in modo da risultare uguali. Nel secondo caso invece si introduce una sola incognita per ogni interse-

zione, variabile che rappresenta la pressione nel punto d'incontro. Per ogni frattura coinvolta inoltre si tiene traccia del grado di libertà in cui avviene l'intersezione. Questo perchè, come vedremo, la condizione al contorno deve essere imposta solo del grado di libertà dell'estremo libero della frattura.

3.5 Le Classi `BC` e `BCHandler`

La classe `BC` ci permette di imporre le condizioni al bordo in ogni singola frattura. Per ogni frattura è necessario imporre due condizioni al contorno.

Listing 3.5: Classe `BC`

```
class BC
{
 public:

    enum
    {
        DIRICHLET_BOUNDARY_NUM = 40,
        NEUMANN_BOUNDARY_NUM = 50
    };

    // Costruttore
    BC ( getfem::mesh& mesh,
         const std::string& MeshType,
         const sizeVector_Type DOFs,
         const ElementDimension& dimension = MEDIUM );

 private:

    getfem::mesh_fem M_meshFEM;

    sizeVector_Type M_dirichlet;
    sizeVector_Type M_neumann;
    sizeVector_Type M_extBoundary;
};
```

Quando si considera una frattura priva di intersezioni, o con solo intersezioni di tipo *Cross*, le condizioni al contorno vengono imposte rispettivamente sul primo e l'ultimo grado di libertà della mesh.
Nel caso della biforcazione, invece, bisogna accertarsi di imporre le condizioni al bordo solamente nell' estremo libero, ovvero quello in cui non sia presente alcuna intersezione. Per far questo, quando costruiamo l'insieme delle intersezioni, per ogni frattura salviamo un vettore contenente l'indice dei gradi di libertà in cui c'è un'intersezione.

La classe `BCHandler` viene usata per gestire le condizioni al contorno sulle fratture.

Capitolo 4

Classi per la gestione dell'intersezione

Nel modello che abbiamo preso in considerazione noi le fratture hanno una rappresentazione in $\mathbb{R}$ e, nel caso della *Biforcazione*, hanno un unico punto in comune. Nella rappresentazione 2D, invece, la regione in comune è un triangolo.

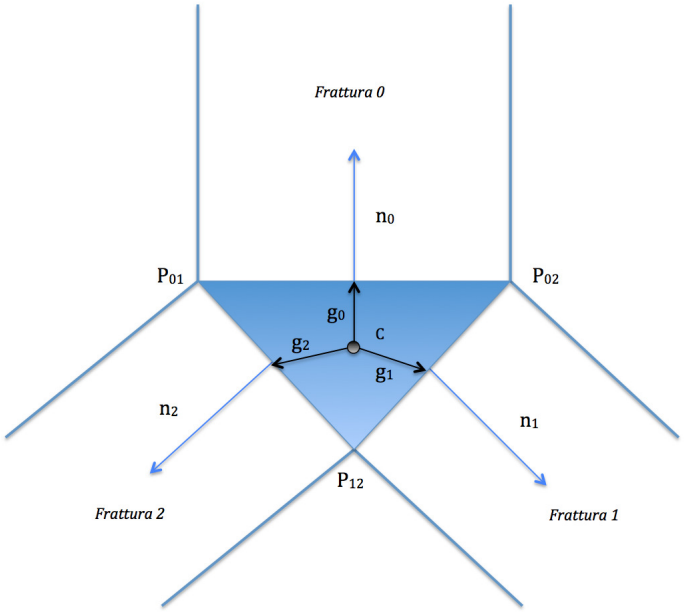

Figura 4.1: Struttura della biforcazione 2D

Il nostro modello trascura ovviamente delle informazioni, ma nel caso di fratture con uno spessore sufficientemente piccolo rispetto a quelle del dominio, può essere considerato una buona approssimazione della realtà.

Per ricavare le condizioni d'interfaccia nel caso della biforcazione per modello ridotto passiamo per la rappresentazione in $\mathbb{R}^2$. Nel caso 2D le fratture hanno uno spessore e possiamo delimitare un triangolo di intersezione unendo i punti di incontro dei loro bordi.
Presi $\mathbf{N}$ il vettore delle normali ai lati del triangolo, K_I la matrice di permeabilità all'intersezione e una costante t possiamo definire:

$$S = t\,diag(\mathbf{N}\ K_I\ \mathbf{N}^T)$$

Definite ora $\mathbf{C}$ matrice dei vettori che uniscono il baricentro, punto C, del triangolo con il punto medio di ogni lato e P_c matrice di proiezione sullo spazio nullo della matrice di base per $\mathbf{C}$, possiamo trovare la matrice di trasmissibilità:

$$T = \tfrac{1}{\mathbf{I}}(\mathbf{N}\ K_I\ \mathbf{N}^T) + P_c\ \mathbf{S} P_c$$

Le variabili con cui ci troviamo a lavorare sono:

- $\mathbf{u}$: vettore dei flussi
- p_I : pressione nel punto di intersezione delle assi delle tre fratture;
- π: vettore delle pressioni nel punto di incontro fra l'asse della frattura i-esima con un lato del triangolo;

e le condizioni che dobbiamo porre all'intersezione sono:

$$\begin{cases} \mathbf{u} - p_I T \mathbf{1}_3 + T\Pi = 0 \\ \\ \sum_{k=0}^{2} u_k = p_I - \pi \end{cases}$$

Queste sono le condizioni d'interfaccia che verranno imposte nel punto di intersezione delle fratture.

In questa parte di codice per gestire sia il calcolo delle matrici introdotte precedentemente, che i punti e la geometria del triangolo di intersezione, abbiamo usato la libreria matematica *Eigen*.

Note le basi teoriche possiamo passare a descrivere le principali classi del codice che definiscono la struttura e le proprietà della biforcazione in uno spazio bidimensionale.

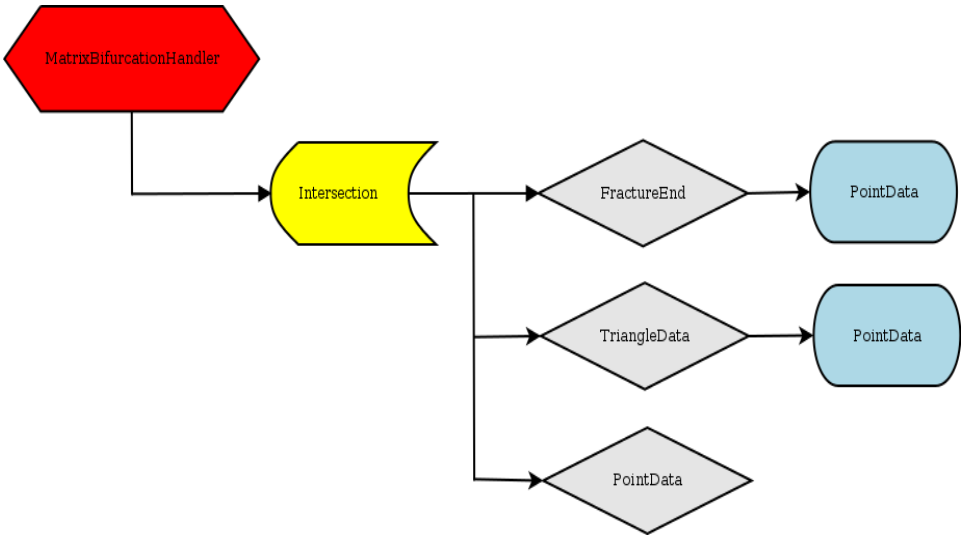

Figura 4.2: Inclusione tra le varie classi che costituiscono il triangolo di intersezione

4.1 La Classe `MatrixBifurcationHandler`

La classe `MatrixBifurcationHandler` contiene le matrici associate a una biforcazione e implementa i metodi per calcolarle. Il fine ultimo è il calcolo della matrice di trasmissibilità T.
I campi fondamentali della classe sono:

- `M_intersection`: di tipo Intersection, contiene tutte le informazioni base per poter ricavare le matrici;
- `M_T`: matrice T di trasmissibilità necessaria per l'imposizione delle condizioni di interfaccia.

Listing 4.1: Classe `Intersection`

```
1  class MatrixBifurcationHandler
2  {
3   public:
4          MatrixBifurcationHandler( const GetPot& dataFile,
5                                    const std::string& section = "mediumData/",
6                                    const std::string& subsection = "darcy/");
7
8          void setMatrices ( FracturePtrContainer_Type& fractures );
9
10         [ ... ]
11
12         void computeT( scalar_type t=6.0 );
```

```
13
14          [ ... ]
15
16  private:
17
18          Matrix2d M_K;
19          Intersection_Type M_intersection;
20          Matrix32 M_N;
21          Matrix32 M_C;
22          Matrix32 M_Qc;
23          Matrix3d M_Pc;
24          Matrix3d M_T;
25  };
```

4.2 La Classe `Intersection`

La class `Intersection`, definita in `Geometry.h`, contiene tutte le informazioni relative ad una biforcazione in uno spazio 2D necessarie per il calcolo delle matrici associate al triangolo di intersezione. I campi fondamentali della classe sono:

- `M_intersection`: di tipo PointData, rappresenta il punto di intersezione;
- `M_tangents`: vettore contenente le tangenti alle fratture coinvolte;
- `M_normals`: vettore contenente le normali alle fratture coinvolte;
- `M_intersectionTriangle`:di tipo TriangleData, rappresenta il triangolo di intersezione, i cui vertici sono i punti d'incontro dei bordi delle fratture.

Listing 4.2: Classe `Intersection`

```
1  class Intersection
2  {
3   public:
4           [ ... ]
5
6           void setIntersection( FracturePtrContainer_Type& M_FracturesSet );
7
8           [ ... ]
9   private:
10
11          [ ... ]
12
13          PointData M_intersection;
14
```

```
15          Vector2d M_tangents [ 3 ];
16          Vector2d M_normals [ 3 ];
17
18          TriangleData M_intersectionTriangle;
19  };
```

4.3 Classe `PointData` e Classe `TriangleData`

Queste classi rappresentano un punto e un triangolo come grandezze geometriche. Il punto geometricamente viene rappresentato come una grandezza zero dimensionale con due variabili, una x e una y. Un triangolo è invece rappresentato geometricamente dai punti che lo costituiscono.
La classe `PointData` ci permette di definire un punto e di poterlo manipolare con le operazioni di somma, differenza e prodotto.
La classe `TriangleData` ci permette di definire un triangolo e di calcolarne l'area. Inoltre sono stati implementati i metodi necessari per la costruzione del triangolo di intersezione.

Capitolo 5

Soluzione del problema numerico

Passiamo ora a descrivere le principali classi del codice per la soluzione del problema numerico.

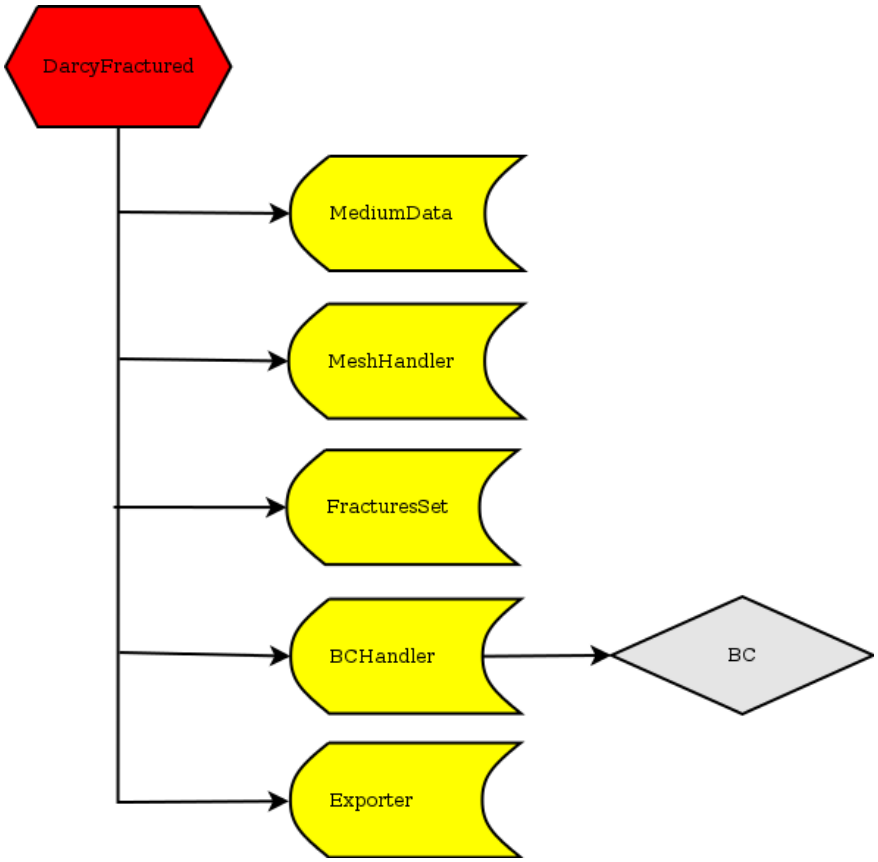

Figura 5.1: Inclusione tra le varie classi per la soluzione del problema numerico

5.1 La Classe `DarcyFracture`

La classe `DarcyFracture` permette di definire il problema di Darcy su ogni frattura, imporre le condizioni per le eventuali intersezioni, assemblare il sistema algebrico globale $Ax = b$ del problema e risolverlo.

I campi fondamentali della classe sono:

- `M_globalMatrix`: è una matrice a blocchi sparsa rappresentata i GETFEM da un *boost::shared_ptr* ⟨ *gmm::row_matrix* ⟨ *sparseVector_Type* ⟩⟩;
- `M_globalRightHandSide`: vettore del termine noto;
- `M_velocityAndPressure`: vettore soluzione delle velocitá e della pressione.

Listing 5.1: Classe `DarcyFracture`

```
1  class DarcyFractured
2  {
3   public:
4      DarcyFractured ( const MediumDataPtr_Type& medium,
5                       const MeshHandlerPtr_Type& mesh,
6                       const BCHandlerPtr_Type& bcHandler,
7                       const FracturesSetPtr_Type& fractures,
8                       const ExporterPtr_Type& exporter );
9
10     void init ( );
11
12     void assembly ( const GetPot& dataFile );
13
14     void solve ( );
15
16         [ ... ]
17
18  private:
19
20         [ ... ]
21
22     sparseMatrixPtr_Type M_globalMatrix;
23
24     scalarVectorPtr_Type M_globalRightHandSide;
25
26     scalarVectorPtr_Type M_velocityAndPressure;
27
28     [ ... ]
29 };
```

Le funzioni fondamentali della classe sono il metodo `assembly ()` e il metodo `solve ()`.
Il metodo `assembly ()` crea la matrice globale A e il termine noto di destra, utilizzando le funzioni della classe `XFEMOperators`. La matrice globale del sistema ha una struttura a blocchi, dove ad ogni frattura corrisponde un blocco derivante dal sistema **??**. Una volta definite le forme bilineari e i blocchi della matrice corrispondenti alle singole fratture, vanno imposte le condizioni d'interfaccia nel caso di eventuali intersezioni.
Nel caso di un *Cross* si impongono le condizioni di interfaccia in maniera debole tramite le matrici E_2 e E_3, che impongono il salto di velocità nell'intersezione.
Nel caso di una *Bifurcation* si impongono in maniera forte le condizioni di interfaccia 4.
Ipotizzando di avere un ambiente dove tre fratture hanno un intersezione di tipo *Bifurcation* e una di queste si incontra con una quarta creando un intersezione di tipo *Cross*, la matrice globale avrà il seguente volto:

$$\begin{bmatrix}
A_0 & B_0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
B_0^T & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & A_1 & B_1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & B_1^T & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & A_2 & B_2 & 0 & 0 & E_2 & 0 \\
0 & 0 & 0 & 0 & B_2^T & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & A_3 & B_3 & 0 & E_3 \\
0 & 0 & 0 & 0 & 0 & 0 & B_3^T & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & App_0 & App_1 \\
0 & 0 & 0 & 0 & E_2^T & 0 & E_3^T & 0 & 0 & 0
\end{bmatrix}.$$

Infine il metodo `solve ()` risolve il problema e esporta i risultati ottenuti in formato *vtk* per entrambe le incognite.

Capitolo 6

Risultati numerici e validazione del modello

Si può subito vedere che l'andamento della pressione e della velocità rispettano la legge di Darcy. Inoltre per validare il modello abbiamo implementato e risolto delle situazioni campione specifiche con l'ausilio di `FreeFem++` che permette la costruzione di domini in uno spazio bidimensionale. Per ogni prototipo sono state effettuate tre simulazioni, diminuendo ad od ogni passo lo spessore delle fratture di un ordine di grandezza. Per ogni frattura abbiamo messo a confronto il valore della pressione nel DOF di intersezione, per il modello monodimensionale, con l'integrale di pressione lungo il lato del triangolo di intersezione mediato sulla sua lunghezza, per `FreeFem++`. Riportiamo inoltre il valore medio della pressione nel punto di intersezione ponendolo a paragone con il valore dell'integrale di pressione sul triangolo di intersezione mediato per l'area di quest'ultimo.
Vediamo ora i risultati ottenuti in alcuni casi significativi.

6.1 Biforcazione semplice, 3 fratture

Consideriamo una biforcazione generata dall'incontro di tre fratture in uno dei loro punti estremi. Confronto valori della pressione nel DOF di intersezione per il modello 2:

	`Code`	`FreeFem++`		
Frattura	-	1E-01	1E-02	1E-03
0	0.000594764	0.372412	0.0674785	-0.000186152
1	-0.0013522	0.244632	0.0553189	-0.000189784
2	0.00225249	0.323944	0.0627701	-0.000187243

	Code	FreeFem++
Pressione Media	0.0004983513	-0.000187683

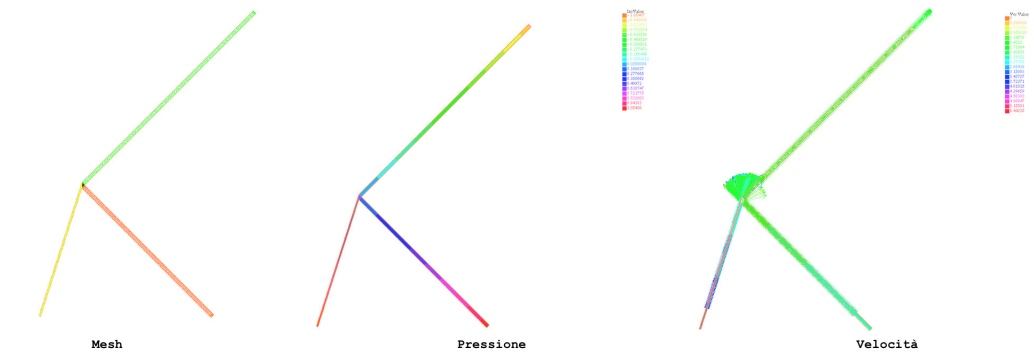

Figura 6.1: Risultati FreeFem++ biforcazione - spessore ordine 1E-02

6.2 Doppia biforcazione e cross, 6 fratture

6.3 Doppia biforcazione verticale, 5 fratture

6.4 Caso limite

Biforcazione con due fratture

6.5 Conclusioni

Bibliografia

[1] Peter Bastian. Numerical computation of multiphase flows in porous media, 1999.

[2] Franco Brezzi, Konstantin Lipnikov, and Valeria Simoncini. A family of mimetic finite difference methods on polygonal and polyhedral meshes, March 17, 2005.

[3] Luca Formaggia. A possible model for bifurcation in 2d fracture networks. January 27, 2014.

[4] Luca Formaggia, Alessio Fumagalli, Anna Scotti, and Paolo Ruffo. A reduced model for darcy's problem in networks of fractures, 2013.