ALMA MATER STUDIORUM

UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E DI ARCHITETTURA

CORSO DI LAUREA MAGISTRALE IN

INGEGNERIA PER L'AMBIENTE E IL TERRITORIO

Linee di flusso per modelli numerici discretizzati con griglie 3D non strutturate: calcolo e visualizzazione sviluppati in ambiente MatLab

Tesi di laurea in Modelli Numerici per la Geoingegneria M

Relatore

Chiar. mo Prof. Villiam Bortolotti

Correlatore

Chiar. mo Prof. Stefano Bonduà

Candidato

Matteo Scapolo

Anno Accademico 2016/2017 – Sessione I

Sommario

In questo lavoro di tesi è stato creato uno strumento informatico sviluppato in ambiente MatLab[1], per il calcolo e la visualizzazione delle traiettorie/linee di flusso del moto di un fluido in un mezzo poroso a partire da simulazioni numeriche ottenute con griglie non strutturate.

Si è ipotizzato che il moto fosse permanente: tale ipotesi permette di eliminare la distinzione fra traiettorie e linee di flusso.

Inizialmente ideato per agire su domini modellizzati con griglie strutturate il codice ha prima implementato il metodo suggerito da Pollock[2]. L'utilizzo delle griglie strutturate si è però dimostrato limitante in termini di domini modellizzabili e di qualità della modellazione, si è deciso quindi di sostituire il metodo di Pollock e concentrarsi sulle griglie non strutturate.

Il nuovo algoritmo implementato è stato quello proposto da Painter, Gable e Kelkar[3] che, operando in due fasi, prima ricostruisce il campo di velocità del fluido nel dominio indagato e poi calcola le traiettorie. Tale algoritmo permette di agire su griglie non strutturate, dove nessun vincolo è posto riguardo alla forma e al numero di facce degli elementi che la compongono.

Il codice gestisce il campo di velocità in due distinte modalità: una che lo crea a partire dai risultati di una simulazione numerica, l'altra che lo carica direttamente da un file già creato per quello scopo.

Nel caso si utilizzi la prima modalità si suppone che i dati di input provengano da simulazioni eseguite con TOUGH2[4] e che la discretizzazione del dominio sia eseguita mediante tassellatura di Voronoi. Si fa notare però che il codice ha architettura

[1] https://it.mathworks.com/products/matlab.html

[2] Pollock D.W. (1988) Semi-analytical computation of path lines for finite-difference models, Ground Water 26(6), pp: 743–750.

[3] Painter S.L., Gable C.W. e Kelkar S. (2012) *Pathline tracing on fully unstructured control-volume grids*, Computational Geosciences 16(4), pp: 1125–1134, doi:10.1007/s10596-012-9307-1.

[4] http://esd1.lbl.gov/research/projects/tough/
Pruess K., Oldenburg C. e Miridis G. (1999) *TOUGH2 User's Guide, Version 2.0*, Earth Sciences Division, Lawrence Berkeley National Laboratory, University of California, Berkeley, California, U.S.A.

modulare e il caricamento dei dati è opportunamente disaccoppiato dalla parte che si occupa della realizzazione delle linee di flusso. Per cui il codice non è intrinsecamente legato a TOUGH2, mediante opportune modifiche è possibile adattarlo ad altri simulatori, come ad esempio FEHM e PFLOTRAN, senza dover agire sull'algoritmo per il calcolo delle traiettorie.

Nella seconda modalità di caricamento dei dati di input, il codice è totalmente indipendente dal tipo di simulatore utilizzato e dal tipo di discretizzazione del dominio.

Il codice prevede che l'utilizzatore fornisca le posizioni iniziali delle particelle di fluido di cui si vuole calcolare le traiettorie e, una volta calcolate, le presenta in forma grafica e le esporta in opportuni file. Il codice crea infatti un file in formato ASCII per ognuna delle traiettorie studiate, in cui è salvata la successione di coordinate costituenti la traiettoria, con relativo tempo di volo e velocità puntuale stimata.

Al fine di mantenere il codice il più semplice e pulito possibile, sono state create due sue versioni dedicate: P2D specifico per domini bidimensionali e P3D per domini tridimensionali.

P2D e P3D sono stati sottoposti ad un approfondito controllo di verifica e validazione utilizzando casi studio sia bidimensionali che tridimensionali.

Il codice trova applicabilità in un'ampia varietà di contesti, dalla bonifica di siti inquinati alla coltivazione di reservoir petroliferi o geotermici. In particolare questo strumento potrebbe essere di estremo interesse nello specifico ambito della simulazione di reservoir basata sulle linee di flusso ("streamline-based" nella letteratura anglosassone).

Indice

Capitolo I - Introduzione 1
1.1 La discretizzazione del dominio 2
1.1.1 Il modello geologico o concettuale 2
1.1.2 Il modello numerico 3
1.2 Le equazioni di flusso 4
1.2.1 Il flusso monofase 5
1.2.2 Il flusso multifase 6
1.2.3 Il flusso frazionario 8
1.2.4 Il flusso di calore 9
1.3 Le equazioni della simulazione streamline-based 10
1.4 Simulazioni streamline-based 11
1.4.1 Le ragioni del successo 12
1.4.2 Gli ambiti di applicazione 13
1.5 Obiettivo della Tesi 14
1.6 Presentazione dei capitoli 15

Capitolo II - Gli strumenti 17
2.1 MatLab 17
2.2 TOUGH2 20
2.3 VORO2MESH 22

Capitolo III - Creazione linee di flusso - Nozioni teoriche 23
3.1 La strategia applicata 23
3.2 La cinematica dei fluidi 25
3.2.1 Elementi caratteristici del moto: traiettorie, linee di flusso e linee di emissione 26
3.2.2 Tipi di moto: moto stazionario 27
3.3 La discretizzazione del dominio 29
3.3.1 Discretizzazione strutturata e non strutturata 29
3.3.2 La tassellatura di Voronoi 30
3.3.3 La tassellatura di Delaunay 31
3.4 Il metodo di Painter 33
3.4.1 Ricostruzione delle velocità al centro delle celle 34

3.4.2 Interpolazione delle velocità 35

Capitolo IV - Il Codice MatLab 37
4.1 I file di input 37
4.1.1 seed.txt 38
4.1.2 elements.txt 38
4.1.3 faces.txt 40
4.1.4 velocities.txt 41
4.1.5 velocityfield.txt 43
4.2 Le funzioni per importare i dati 44
4.3 P3D 45
4.3.1 Section 1, definizione dei parametri da parte dell'utente 45
4.3.2 Section 2, determinazione del campo di velocità 47
4.3.3 Section 3, generazione della griglia di Delaunay 56
4.3.4 Section 4, calcolo delle traiettorie 57
4.3.5 Section 5, visualizzazione e salvataggio delle traiettorie 60
4.4 P2D 62
4.5 I file di output 62

Capitolo V - Verifica e Validazione 65
5.1 Test bidimensionali 65
5.1.1 Griglia strutturata 66
5.1.2 Griglia non strutturata 68
5.1.3 Tempi di calcolo 69
5.2 Test tridimensionali 70
5.2.1 Griglia strutturata 71
5.2.2 Griglia non strutturata 71
5.2.3 Tempi di calcolo 73
5.2.4 Dominio con barriera quasi totalmente impermeabile 73

Capitolo VI - Conclusioni e futuri sviluppi 77

Appendice A – P3D 79

Appendice B – P2D 95

Bibliografia 113

Sitografia 117

Indice delle figure

Figura 2.1 – Schermata di avvio del codice con MatLab 19
Figura 3.1 – Esempio di tassellatura di Delaunay 31
Figura 3.2 – Esempio di tassellatura che non rispetta il criterio di Delaunay 31
Figura 3.3 – Triangolazioni di Delaunay non univoche 32
Figura 3.4 – Tassellatura di Delaunay 3D 32
Figura 3.5 – Discretizzazione del dominio nel metodo di Painter 33
Figura 4.1 – Opzioni dell'utente di TOUGH2 per fornire i dati geometrici a TOUGH2 39
Figura 4.2 – Organizzazione dei caratteri nel blocco ELEME. 40
Figura 4.3 – Intestazione del file di output di TOUGH2 41
Figura 4.4 – Blocco di dati utili allo script nel file di output di TOUGH2 42
Figure 4.5 – Come si deve presentare il file velocities.txt 42
Figure 4.6 – Come si deve presentare il file velocityfield.txt 43
Figure 4.7 – Come si presenta il file Pathline.txt 63
Figura 5.1 – Schema five spot 65
Figura 5.2 – Soluzione P2D e analitica, schema five spot griglia strutturata a 121 elementi 66
Figura 5.3 – Soluzione P2D e analitica, schema five spot griglia strutturata a 676 elementi 67
Figura 5.4 – Soluzione P2D e analitica, schema five spot griglia non strutturata a 243 elementi 68
Figura 5.5 – Griglia strutturata 3D 70
Figura 5.6 – Soluzione di P3D, griglia strutturata 71
Figura 5.7 – Griglia non strutturata 3D 72
Figura 5.8 – Soluzione di P3D, confronto fra soluzioni fra griglie strutturate e non strutturate .. 72
Figura 5.9 – Test P3D, barriera 74
Figura 5.10 – Soluzione di P3D, dominio con barriera, visuale prospettica 74
Figura 5.11 – Soluzione di P3D, dominio con barriera, sezione 2D nel piano z=0 75
Figura 5.12 – Soluzione di P3D, dominio con barriera, sezione 2D nel piano x=0 75

Indice delle tabelle

Tabella 2.1 – Equation Of State di TOUGH2 41
Tabella 4.1 – Schema riassuntivo delle informazioni fornite in tough2viewer.dat 40
Tabella 5.1 – Tempo di calcolo di P2D con calcolo del campo di velocità 69
Tabella 5.2 – Tempo di calcolo di P2D, campo di velocità già calcolato 69
Tabella 5.3 – Tempo di calcolo di P3D con calcolo del campo di velocità 73
Tabella 5.4 – Tempo di calcolo di P3D, campo di velocità già calcolato 73

Capitolo I

Introduzione

L'Enciclopedia Treccani definisce *sistema* come "*qualsiasi oggetto di studio che, pur essendo costituito da diversi elementi reciprocamente interconnessi e interagenti tra loro e con l'ambiente esterno, reagisce o evolve come un tutto, con proprie leggi generali*" [37].

In ultima analisi, il sistema target oggetto di studio di questa tesi è il giacimento o *reservoir* di idrocarburi, definito a sua volta come un accumulo di idrocarburi, olio e/o gas, in rocce porose e permeabili; la loro presenza è accompagnata sempre anche da acqua e in misura minore di altri componenti chimici. Le varie componenti fluide possono raccogliersi in fasi diverse [13]. In generale, però, per lo sviluppo di questo specifico lavoro, non sarà necessario fare riferimento alle specifiche proprietà dei reservoir di idrocarburi.

Il sistema reservoir è caratterizzato da grande complessità, per il suo studio quindi lo si sostituisce con un sistema immagine detto *modello*. Il modello, specifico del sistema considerato e del problema da risolvere su di esso, è un sistema più semplice, che può essere di tipo fisico o astratto (in genere matematico), che contiene solamente gli elementi del sistema originario ritenuti determinanti per la risoluzione del problema.

Se il modello è rappresentativo del sistema, allora le prove eseguite su di esso forniscono informazioni relative al comportamento del sistema da cui il modello è derivato: questa particolare operatività prende il nome di *simulazione* [7].

L'importanza della simulazione in ambito petrolifero è di immediata comprensione: essa permette di "prevedere" il comportamento "interno" del giacimento in termini di pressioni e saturazioni dei fluidi nel tempo nonché le sue manifestazioni "esterne" in termini di portate di olio, gas e acqua prodotti in funzione dei piani di coltivazione ipotizzati [11].

In questo elaborato si farà riferimento a due diversi approcci di simulazione appartenenti alla branca della *simulazione numerica*, quella che prevede cioè la costruzione di un modello logico-matematico del sistema in esame analizzabile con un elaboratore [36].

Il primo, che individueremo con l'aggettivo "*cell-based*" è l'approccio tradizionale adottato, ad esempio, nei metodi alle differenze finite o agli elementi finiti [18]. Questo approccio prevede la discretizzazione del volume del giacimento, lo si suddivide cioè in elementi di volume detti *celle* o *blocchi* come definiti da una specifica *griglia*. Anche il tempo viene discretizzato in passi di tempo non necessariamente uguali fra loro. Si passa quindi da una rappresentazione continua delle proprietà petrofisiche come pure delle variabili termodinamiche del fluido a una rappresentazione discretizzata che ovviamente dipende dal tipo di discretizzazione adottata: i valori di tutte le grandezze sono supposti costanti all'interno di ciascun blocco [7].

Il secondo approccio, che individueremo con l'aggettivo "*streamline-based*", ha come elemento cardine non più la cella bensì la linea di flusso (*streamline*), lungo cui vengono calcolate saturazioni e componenti delle diverse fasi [18].

In questa Introduzione, per la definizione del problema del moto fluido in un mezzo poroso e delle equazioni che lo rappresentano verrà seguito il lavoro di Matringe in [21].

1.1 La discretizzazione del dominio

1.1.1 Il modello geologico o concettuale

Il primo passo è ottenere un modello di giacimento affidabile, ossia avere una sua rappresentazione geologica accurata e approfondita. Le difficoltà poste dalla posizione difficilmente raggiungibile del giacimento, migliaia di metri al di sotto della superficie, limitano considerevolmente la quantità di dati disponibili per la sua caratterizzazione. L'acquisizione di dati misurati ("*hard*") sulle proprietà della roccia è possibile solamente mediante la perforazione di pozzi; esistono però numerose tecniche basate su metodi geofisici che permettono di ricostruire le proprietà della roccia sulla base di dati

acquisibili dalla superficie. È il caso dei dati sismici ad esempio, mediante misurazioni di velocità e impedenza è possibile ricavare la porosità della roccia. Le tecniche geofisiche attuali permettono di definire le caratteristiche geologiche del reservoir a larga scala, descrivendo quindi la geometria delle varie formazioni e riportando faglie o fratture. Altre caratteristiche a piccola scala, come variazioni locali di porosità o permeabilità, non sono invece indagabili. Ne consegue che tali informazioni, fondamentali per la caratterizzazione del reservoir, devono necessariamente provenire da metodi diversi.

Il modello geologico può essere creato mediante metodi di geostatistica che integrano i dati "hard" provenienti da carote e log di pozzo con i dati "*soft*" ottenuti con le prove geofisiche. Si completano quindi le informazioni a larga scala con quelle a piccola scala fornite in senso statistico ottenendo un modello geologico il più adeguatamente dettagliato.

I reservoir sono in genere rappresentati con griglie composte da elementi abbastanza "piccoli" da poter descrivere accuratamente sia le principali caratteristiche geologiche sia le variazioni delle proprietà della roccia: caratteristiche queste entrambe indispensabili per ottenere modelli affidabili nella riproduzione dei fenomeni di flusso.

1.1.2 Il modello numerico

Il livello di dettaglio dei modelli geologici è molto alto e questo li rende poco adatti per essere utilizzati tal quali nella simulazione del reservoir, basti pensare che un modello geologico può essere composto da griglie con $10^7 – 10^8$ blocchi. Il progresso delle tecniche computazionali e delle capacità dei calcolatori ha con il tempo reso possibile la simulazione diretta su modelli geologici così estesi, ma essa continua a non essere vantaggiosa in termini computazionali.

Le simulazioni di flusso vengono quindi eseguite su modelli (il modello numerico) con griglie di dimensioni minori, $10^5 – 10^6$ blocchi, costruiti ad hoc mediante operazioni di cambiamento di scala (*upscaling*) che permettono di derivare le proprietà del mezzo poroso alla scala più grossolana a partire da quelle alla scala inferiore [14]. Esiste una grande varietà di tecniche di upscaling il cui scopo è quello di mantenere, nonostante la

“compressione” di dati, tutte le caratteristiche del modello geologico funzionali alla simulazione.

La selezione di tali caratteristiche avviene allo stato d’arte attuale mediante l’esecuzione di una simulazione di flusso sul modello geologico compiuta con ipotesi semplificative quali la presenza di un un’unica fase nel reservoir. Questa prima selezione può essere vantaggiosamente eseguita mediante l’impiego delle linee di flusso che forniscono in modo naturale le caratteristiche da mantenere. Alcune caratteristiche a larga scala, come ad esempio le faglie principali, contribuiscono in maniera significativa a caratterizzare il moto; altre invece, come la presenza di determinate stratificazioni sedimentarie, hanno un impatto minimo e sono spesso omesse dal modello numerico. Allo stesso modo, strati altamente permeabili o piccole fratture possono influenzare fortemente lo sviluppo del flusso e devono quindi essere presi in considerazione; al contrario le linee di flusso possono essere totalmente indifferenti ad altre caratteristiche geologiche a piccola scala.

La griglia di simulazione deve essere opportunamente progettata per seguire da vicino le caratteristiche strutturali del modello geologico che hanno forte impatto sul flusso. Gli elementi della griglia più usati sono quelli di forma esaedrica (griglie strutturate), eventualmente distorti per adattarsi alle geometrie del reservoir. Sempre più interesse però viene posto all’uso di griglie non strutturate. Queste infatti riescono a riprodurre fedelmente le complesse geometrie del giacimento con molti meno blocchi di quelli necessari usando volumi esaedrici.

Una volta scelta la griglia che meglio descrive le geometrie del reservoir bisogna associare ad ogni elemento i parametri petrofisici, le condizioni iniziali delle variabili e le condizioni al contorno.

1.2 Le equazioni di flusso

Vengono ora introdotte sinteticamente le principali equazioni che governano il flusso di un fluido all’interno di un mezzo poroso. Si considerano i casi di flusso monofase,

polifase e frazionario, le equazioni che descrivono quest'ultimo caso sono alla base della simulazione streamline-based.

1.2.1 Il flusso monofase

La condizione di flusso monofase assume particolare rilevanza nello sviluppo delle tecniche di simulazione numerica. La prima relazione introdotta è la ben nota legge di Darcy che quantifica la velocità volumetrica $\boldsymbol{u}$ [m/s] come

$$\boldsymbol{u} = -\frac{\boldsymbol{k}}{\mu}\nabla\Phi. \tag{1.1}$$

avendo individuato con $\boldsymbol{k}$ [m^2] il tensore di permeabilità del mezzo poroso, μ [Pa·s] la viscosità dinamica del fluido e Φ [Pa] il potenziale del fluido per unità di volume dello stesso. La formulazione del potenziale è la seguente:

$$\Phi = p + \rho g D + \frac{v^2}{2}\rho. \tag{1.2}$$

dove p [Pa], ρ [kg/m^3], g [m/s^2], D [m] e v sono, rispettivamente, pressione e densità del fluido, la costante di accelerazione gravitazionale, la quota in termini di profondità rispetto al livello di riferimento (datum) e la velocità del fluido.

Nei mezzi porosi la velocità di flusso è sempre molto bassa, il termine dinamico viene quindi trascurato. Vengono in questa fase introduttiva trascurati anche gli effetti gravitativi, per cui il potenziale viene identificato con la pressione del fluido.

L'equazione di continuità che traduce il principio di conservazione della massa può essere scritta come

$$\frac{\partial\, \phi\rho}{\partial\, t} + \nabla\cdot(\rho\boldsymbol{u}) = \rho f. \tag{1.3}$$

avendo individuato la porosità della roccia con il simbolo ϕ. L'estrazione o l'iniezione di fluidi nel sistema viene rappresentata attraverso il simbolo f [m^3/s], che esprime la

portata volumetrica di fluido sempre per unità di volume della roccia. Nel caso di fluido incompressibile la densità è considerata costante, la (1.3) è quindi semplificata in

$$\nabla \cdot \boldsymbol{u} = f. \tag{1.4}$$

che combinata alla legge di Darcy (1.1) produce, in termini di pressione:

$$-\nabla \cdot \frac{\boldsymbol{k}}{\mu} \nabla p = f. \tag{1.5}$$

1.2.2 Il flusso multifase

Si consideri ora un flusso di due fasi fluide immiscibili e incompressibili. Si individui la fase bagnante, tipicamente l'acqua, con il pedice *w* (da *wetting*) e la fase non bagnante, tipicamente l'olio, con il pedice *o*. Supposto il volume poroso completamente saturo la relazione che lega le saturazioni delle due fasi è:

$$S_w + S_o = 1. \tag{1.6}$$

Le pressioni delle singole fasi possono essere invece collegate fra loro mediante l'introduzione della pressione capillare p_c, risulta quindi essere:

$$p_c(S_w) = p_o - p_w. \tag{1.7}$$

Trascurando poi gli effetti della pressione capillare la relazione (1.7) diviene:

$$p_w = p_o = p. \tag{1.8}$$

Mantenendo l'ipotesi di fluidi incompressibili per ognuna delle fasi l'equazione della conservazione della massa può essere scritta come:

$$\phi \frac{\partial S_w}{\partial t} + \nabla \cdot \boldsymbol{u_w} = f_w, \tag{1.9}$$

$$\phi \frac{\partial S_o}{\partial t} + \nabla \cdot \boldsymbol{u_o} = f_o. \tag{1.10}$$

Allo stesso modo può essere riscritta la legge di Darcy generalizzandola al caso multifase introducendo le permeabilità relativa k_r:

$$\boldsymbol{u_w} = -\frac{k_{rw}}{\mu_w} \boldsymbol{k}\, \nabla p, \tag{1.11}$$

$$\boldsymbol{u_o} = -\frac{k_{ro}}{\mu_o} \boldsymbol{k}\, \nabla p. \tag{1.12}$$

Introducendo l'espressione della mobilità

$$\lambda_w = \frac{k_{rw}}{\mu_w} \tag{1.13}$$

le relazioni (1.11) e (1.12) diventano:

$$\boldsymbol{u_w} = -\lambda_w \boldsymbol{k}\, \nabla p, \tag{1.14}$$

$$\boldsymbol{u_o} = -\lambda_o \boldsymbol{k}\, \nabla p. \tag{1.15}$$

Sostituendo infine le relazioni (1.14) e (1.15) appena ottenute nelle (1.9) e (1.10) si ottengono rispettivamente

$$\phi \frac{\partial S_w}{\partial t} - \nabla \cdot \lambda_w \boldsymbol{k}\, \nabla p = f_w, \tag{1.16}$$

$$\phi \frac{\partial S_o}{\partial t} - \nabla \cdot \lambda_o \boldsymbol{k}\, \nabla p = f_o, \tag{1.17}$$

anche dette equazioni di diffusività relative alla singola fase.

1.2.3 Il flusso frazionario

Adottando la formulazione di flusso frazionario le equazioni di flusso multifase possono essere notevolmente semplificate, si ottiene così una equazione in termini di pressione e una in termini di saturazione per ognuna delle fasi presenti nel sistema. Introducendo la velocità totale

$$\boldsymbol{u} = \boldsymbol{u_w} + \boldsymbol{u_o} \tag{1.18}$$

e sostituendo al suo interno le relazioni (1.14) e (1.15) si ottiene:

$$\boldsymbol{u} = -(\lambda_w + \lambda_o)\boldsymbol{k}\,\nabla p. \tag{1.19}$$

Definita quindi la mobilità totale del fluido come

$$\lambda = \lambda_w + \lambda_o \tag{1.20}$$

la (1.19) diviene:

$$\boldsymbol{u} = -\lambda\,\boldsymbol{k}\,\nabla p. \tag{1.21}$$

Sommando la (1.9) e la (1.10), tenendo conto della (1.6) si può esprimere l'equazione di continuità come

$$\boldsymbol{\nabla} \cdot \boldsymbol{u} = f_w + f_o = f. \tag{1.22}$$

Combinando la (1.21) con la (1.22) si ottiene, in termini di pressione, l'equazione

$$-\boldsymbol{\nabla} \cdot (\,\boldsymbol{k}\,\nabla p) = f \tag{1.23}$$

che rappresenta la parte del problema relativa al flusso.

Indicando ora con φ_w la funzione di flusso frazionario per la fase bagnante si può scrivere:

$$\varphi_w = \frac{\lambda_w}{\lambda}. \tag{1.24}$$

La velocità della fase bagnante rispetto alla velocità totale sarà quindi:

$$\boldsymbol{u_w} = \varphi_w \boldsymbol{u}. \tag{1.25}$$

Sostituendo la (1.25) nella (1.9) si ottiene:

$$\phi \frac{\partial S_w}{\partial t} + \nabla \cdot (\varphi_w \boldsymbol{u}) = f_w \tag{1.26}$$

che pone in relazione la saturazione della fase bagnate con la velocità totale del fluido; con la (1.26) viene descritta la parte di problema relativa al trasporto di massa. Relazioni analoghe possono essere scritte per la fase non bagnante.

Questa formulazione che divide il problema di flusso dal problema di trasporto è particolarmente vantaggiosa: essa permette infatti di usare metodi numerici diversi per risolvere separatamente le due equazioni che hanno proprietà matematiche molto diverse fra loro.

1.2.4 Il flusso di calore

Nel caso in cui fossero modellizzati reservoir non isotermi (come nel caso di reservoir geotermici) sarà necessario impostare anche una equazione che descriva il flusso di calore C [W] trasferito per conduzione e convezione:

$$C = -\lambda \nabla T + \sum_{\beta} h_\beta F_\beta \tag{1.27}$$

avendo individuato con λ [W/m·K] la conducibilità termica, con T[K] la temperatura e con h_β [J/kg] e F_β [kg/s] l'entalpia specifica e il flusso della generica fase β.

1.3 Le equazioni della simulazione streamline-based

Una strategia già più volte usata prevede l'uso combinato di metodi cell-based e di metodi streamline-based [23]: una descrizione dei principali passaggi che la compongono è qui riportata.

Sono mantenute per semplicità le ipotesi introdotte in precedenza, si considera infatti un flusso bifase e vengono trascurati gli effetti dovuti alle forze capillari e alla gravità. Il metodo streamline si basa sulla formulazione introdotta nel §1.2 in termini di flusso frazionario. Posto $f = 0$ l'equazione (1.26) descrivente il problema di trasporto per la fase bagnante può essere scritta come:

$$\phi \frac{\partial S_w}{\partial t} + \nabla \cdot (\varphi_w \boldsymbol{u}) = 0 \tag{1.28}$$

Individuata con s la lunghezza della linea di flusso, e sostituita l'identità

$$\nabla \cdot \boldsymbol{u} \equiv \|\boldsymbol{u}\| \frac{\partial}{\partial s} \tag{1.29}$$

nella relazione (1.28), il problema di trasporto può essere riformulato in termini monodimensionali

$$\phi \frac{\partial S_w}{\partial t} + \|\boldsymbol{u}\| \frac{\partial \varphi_w}{\partial s} = 0. \tag{1.30}$$

Si introduce ora il concetto di tempo di volo, individuato con il simbolo τ e l'acronimo TOF (nella letteratura anglosassone, Time Of Flight). Il tempo di volo è definito come il tempo impiegato da una particella per percorrere una certa distanza o raggiungere una determinata posizione, esso può essere legato alla lunghezza s e la velocità totale mediante

$$\frac{\partial \tau}{\partial s} = \frac{\phi}{\|\boldsymbol{u}\|}. \tag{1.31}$$

Sostituendo la (1.31) nella (1.30) il problema di trasporto di massa risulta governato dalla semplice equazione monodimensionale lungo il percorso della linea di flusso

$$\frac{\partial S_w}{\partial t} + \frac{\partial \varphi_w}{\partial t} = 0. \tag{1.32}$$

La strategia riportata prevede prima la soluzione del problema di flusso e poi la soluzione di quello di trasporto modificato nella notazione appena ricavata.

Dapprima si ottiene la soluzione in termini di pressione mediante l'utilizzo di un simulatore cell-based, da questa vengono poi estrapolate le linee di flusso. Il problema di trasporto viene poi risolto mediante la relazione (1.32) che fornisce le saturazioni lungo le linee di flusso appena calcolate. Ne risulta quindi la fondamentale importanza di un'accurata determinazione delle linee di flusso in termini di coordinate e tempo di volo.

La ricostruzione delle linee di flusso avviene in due passaggi.

Il primo è la ricostruzione del campo di velocità a partire dai flussi associati agli elementi usati dal simulatore cell-based. Questo passaggio ovviamente dipende dalla discretizzazione del dominio utilizzata dal simulatore e si complica in presenza di griglie non strutturate.

Il secondo passaggio invece prevede la determinazione delle linee di flusso in termini di coordinate e tempo di volo a partire dal campo di velocità appena ricostruito [21].

1.4 Simulazioni streamline-based

I punti di forza e gli impieghi della simulazione streamline-based sono documentati da una vasta produzione scientifica. Ad esempio, in ambito petrolifero si citano i contributi di King e Datta-Gupta [17], il testo è datato ma offre una panoramica chiara ed

esauriente sugli elementi chiave del metodo, e per un testo più recente, Al-Najem et al. [1] che fornisce una rassegna di più di 200 articoli sull'argomento.
La simulazione streamline-based si sta dimostrando uno strumento potente anche in ambito idrologico per l'analisi del trasporto di inquinanti che interagiscono fisicamente e chimicamente con rocce e sedimenti, si citano sull'argomento i testi di Crane et al. [12] e Finkel et al. [15].

1.4.1 Le ragioni del successo

La caratteristica forse più utile della simulazione streamline-based è legata al potere insito nella rappresentazione grafica delle linee di flusso che definiscono lo schema globale di flusso nel mezzo poroso [26]. Le linee di flusso, infatti, offrono una visione immediata di come i pozzi, la geometria del reservoir e la sua eterogeneità interagiscano nel definire come il flusso evolve dai pozzi di iniezione a quelli di emungimento. Questo approccio visivo al problema permette di evidenziare eventuali incongruenze nel comportamento dei fluidi all'interno del reservoir derivanti, ad esempio, dall'utilizzo di un modello geologico "sbagliato".
Un ulteriore vantaggio della simulazione streamline-based rispetto ai metodi tradizionali (simulazioni alle differenze finite, agli elementi finiti, etc.) risiede nella sua intrinseca efficienza computazionale. A parità di domini indagati la simulazione streamline-based necessita di inferiori tempi di calcolo e di computer con una minor quantità di memoria.
La simulazione streamline-based offre inoltre informazioni che non sarebbero altrimenti disponibili con l'uso di approcci tradizionali. È possibile infatti determinare quali pozzi di iniezione, o quali parti dell'acquifero, stanno alimentando determinati pozzi di emungimento: in questo modo è possibile risalire alle cause che provocano, ad esempio, un alto *watercut* in un determinato pozzo. All'inverso è possibile determinare come si ripartisce il volume iniettato in un particolare pozzo fra i vari pozzi di emungimento che sta alimentando.
Un'altra possibilità data dalla simulazione streamline-based è quella di poter determinare per ogni singolo pozzo di iniezione o emungimento l'effettivo volume di reservoir da esso influenzato; ciò è possibile perché ogni cella attraversata dalle linee di

flusso che hanno origine o fine in quel particolare pozzo fa parte del suo volume di influenza [26].

1.4.2 Gli ambiti di applicazione

Come già anticipato la simulazione streamline-based viene correntemente utilizzata in ambito idrologico e in ambito petrolifero. Le applicazioni in ambito petrolifero sono diverse, le più interessanti riguardano probabilmente l'history matching e la determinazione dei *well allocation factors* e *well pore volumes* [26].

L'history matching, cioè la riproduzione da parte del modello della storia passata del reservoir, è un problema inverso noto per essere fortemente non lineare e mal posto; esso produce infatti una serie di soluzioni non uniche. L'ottenimento di un modello validato, che riproduce quindi la storia passata, richiede in genere numerose simulazioni. Uno dei vantaggi principali della simulazione streamline-based risiede, come visto, nella sua efficienza computazionale; ciò permette di eseguire a parità di tempo un numero maggiore di simulazioni rispetto a quelle eseguibili con un simulatore numerico tradizionale.

L'utilità della simulazione streamline-based nell'ambito dell'history matching non si ferma però a una maggiore efficienza e velocità computazionale. Essa in particolare è stata usata per ottenere analiticamente i coefficienti di sensitività e la evidenziazione delle regioni di reservoir associate ai pozzi in cui l'history matching fallisce. Questo permette in seguito di poter cambiare puntualmente le caratteristiche associate ai blocchi della griglia di quelle regioni.

La simulazione streamline-based permette inoltre di ottenere due dati che si prospettano come fra i più utili nel prossimo futuro nell'ambito del recupero d'olio secondario e terziario mediante l'iniezione in pozzo di acqua o gas. Trattasi dei fattori di allocazione di pozzo (well allocation factors), che quantificano la quota parte di flusso in un particolare pozzo produttore che può essere imputato ai pozzi iniettori, e i volumi porosi di pozzo (well pore volumes), che sono invece i volumi di reservoir associati ad ogni singolo pozzo.

Conoscere i fattori di allocazione fra le diverse coppie di pozzi di iniezione/ emungimento, determinati finora con metodologie empiriche, è il punto di partenza per il bilancio puntuale dei volumi iniettati ed emunti e dell'efficienza di un pozzo di iniezione.

La simulazione streamline-based può essere anche utilizzata per avere una migliore caratterizzazione dell'efficienza dei pozzi di produzione. Per quest'ultimi un indicatore di efficienza è già presente ed è il watercut, maggiore è quest'ultimo minore è l'efficienza del pozzo produttore. Ma poiché le linee di flusso permettono di determinare il volume poroso associato ad ogni singolo pozzo si possono quindi fare confronti incrociati fra quantità di olio prodotto e la saturazione media di olio del mezzo poroso associato ad ogni pozzo produttore. Viene quindi definito un nuovo criterio di efficienza per pozzi produttori, quelli che a fronte di basse saturazioni d'olio sono caratterizzati da un minore watercut. La classificazione dei pozzi in base a quest'ultimo criterio di efficienza può essere molto utile per la pianificazione della manutenzione, come interventi di workover o sidetracking [26].

1.5 Obiettivo della Tesi

L'obiettivo della Tesi è la creazione di uno strumento (nello specifico un software) che calcoli e permetta di visualizzare le traiettorie/linee di flusso di un fluido in moto stazionario all'interno di un mezzo poroso a partire dai dati simulati con un modello cell-based caratterizzato da griglia non strutturata bi o tridimensionale.

L'ambiente di sviluppo per realizzare questo strumento è stato individuato, per le sue potenzialità computazionali e grafiche, in MatLab.

Ci si prefigge di rendere questo strumento facilmente accessibile anche a utenti non esperti di MatLab, di limitare le operazioni di preprocessing sui dati di input e soprattutto di renderlo potenzialmente accoppiabile a qualsiasi simulatore cell-based.

1.6 Presentazione dei capitoli

Nel Capitolo II verranno presentati i programmi utilizzati nel corso del lavoro di tesi. Particolare attenzione sarà rivolta a MatLab [32] ma ampio spazio sarà dedicato anche a TOUGH2 [30], sulle cui simulazioni si fonda tutto il lavoro relativo alla determinazione del campo di velocità; verrà infine introdotto VORO2MESH [5] che fornisce altri dati necessari al codice nel caso di griglie non strutturate.

Nel Capitolo III vengono forniti alcuni concetti teorici utili alla comprensione del lavoro svolto, si parlerà dapprima di cinematica dei fluidi per passare poi all'argomento della discretizzazione del dominio, da ultimo sarà approfondito il metodo di calcolo delle traiettorie vero e proprio.

Nel Capitolo IV è riportato l'intero codice e le operazioni codificate sono spiegate singolarmente.

Il codice, nelle due versioni bi e tridimensionale, viene testato su opportuni domini: i risultati ottenuti nei test sono riportati e commentati nel Capitolo V.

Chiude il Capitolo VI dove vengono riassunte le conclusioni di questo lavoro di tesi e si propongono gli ulteriori sviluppi a cui si intende sottoporre il codice.

Capitolo II

Gli strumenti

L'ambiente di sviluppo utilizzato per la creazione del codice è stato MatLab, i dati di input provengono invece dal simulatore cell-based multifase – multicomponente non isotermo TOUGH2 e dal software di preprocessing specifico per TOUGH2 VORO2MESH che permette di creare griglie 3D non strutturate.

Nel seguito del Capitolo questi programmi verranno brevemente presentati illustrandone potenzialità e ambiti di applicazione.

2.1 MatLab

MatLab [3][9][27][32], il cui nome deriva dall'unione delle parole MATrix LABoratory, è un software in grado di eseguire operazioni di tipo numerico, grafico e di programmazione; tale nome individua inoltre lo specifico linguaggio di programmazione utilizzato nel software.

Sviluppato all'inizio degli anni '80, MatLab è stato originalmente scritto per fornire facile accesso a software come LINPACK e EISPACK, che rappresentavano insieme la punta del progresso software per il calcolo matriciale.

Nel corso degli anni MatLab si è evoluto anche e soprattutto grazie agli input provenienti da molti utenti. La sua struttura, infatti, è facilmente implementabile con dei "pacchetti" aggiuntivi denominati toolbox.

L'elemento cardine di questo programma, come si può intuire dal suo nome, è la matrice; vettori e anche elementi scalari come i numeri sono considerati come particolari matrici dalle opportune dimensioni. Tali matrici non necessitano di essere dimensionate e ciò permette la risoluzione più agevole di problemi di calcolo tecnici espressi appunto in notazioni matriciali o vettoriali; gli algoritmi utilizzati risultano

essere più semplici e snelli rispetto a quelli che sarebbero necessari in un programma in linguaggio imperativo come il C o il FORTRAN.

Un altro considerevole punto di forza di MatLab è la vasta libreria di funzioni matematiche built-in: tramite queste è possibile compiere con un solo comando operazioni elementari, come somma e moltiplicazione, o complesse, come il calcolo di matrici inverse e il calcolo di autovalori. La libreria di funzioni utilizzabili può essere ulteriormente espansa con i sopraccitati toolbox; ne esiste una vastissima gamma che contiene strumenti creati appositamente per i più disparati campi di applicazione che vanno, per citarne solo alcuni, dalla statistica alla robotica.

In questa trattazione in particolare sono stati utilizzati alcuni toolbox relativi all'elaborazione dell'immagine come, ad esempio, l'Image Processing Toolbox. Ciò ha permesso di aumentare ulteriormente le già considerevoli potenzialità grafiche di MatLab, che permettono di creare immagini a colori e grafici bi e tri-dimensionali.

Il programma MatLab e relativi toolbox sono stati usati nella versione R2016a ottenuta mediante la licenza campus dell'Ateneo di Bologna.

MatLab può eseguire comandi singoli se il dato richiesto è prodotto da una semplice operazione. Quando però l'ottenimento del risultato finale necessita di più passaggi è conveniente raggruppare i singoli comandi in un programma che nel seguito sarà riferito come *codice*.

Avviando il codice con MatLab la schermata che appare è mostrata in Figura 2.1.

La suddivisione degli spazi della schermata, nella conformazione standard offerta da MatLab 2016a, offre la *Current Folder Window* in alto a sinistra, il *Workspace* in basso a sinistra, l'*Editor* di file sulla destra e sotto la *Command Window*. Si precisa che comunque la disposizione di questi vari elementi varia a seconda della versione di MatLab installata e può essere personalizzata. La Current Folder Window mostra tutti i file contenuti nella cartella da dove è stato aperto il codice, per il corretto funzionamento dello stesso in essa dovranno essere presenti tutti i file di input e file MatLab di importazione dati descritti in seguito.

La Workspace Window contiene le variabili utilizzate dal codice e i valori ad esse associati.

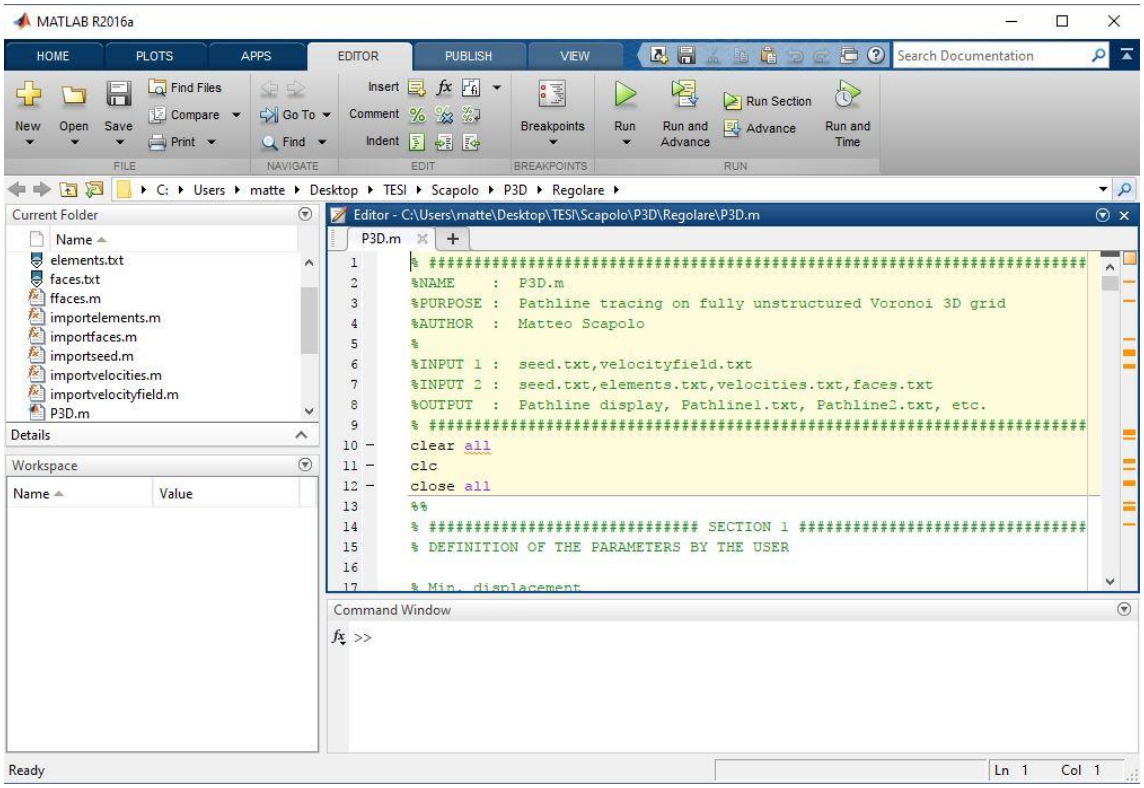

Figura 2.1 - Schermata di avvio del codice con MatLab

.

Aprendo uno specifico file .m, estensione caratteristica di MatLab, si apre l'Editor di tale file. In questa sezione si possono apportare modifiche al codice, cosa che sarà richiesta in seguito per calibrarlo alla specifica simulazione.

Nell'ultima finestra, la Command Window, non è richiesta nessuna operazione da parte dell'utente del codice. In generale in essa MatLab può essere utilizzato in modo interattivo mediante lo strumento >> chiamato *prompt.* Comandi MatLab o espressioni possono essere digitati dopo il prompt e MatLab risponderà immediatamente con il risultato. La stampa nella Command Window della maggior parte di questi risultati è stata soppressa nel codice, le uniche informazioni che saranno mostrate serviranno all'utente per verificare lo stato di avanzamento delle operazioni di calcolo.

2.2 TOUGH2

TOUGH2 [25][29][30] è un programma di simulazione numerica per flussi non isotermi, multicomponente e multifase in mezzi fratturati e porosi; è il più famoso fra i codici della famiglia TOUGH, acronimo per Trasport Of Unsaturated Groundwater and Heat. Le applicazioni di TOUGH2 sono innumerevoli: è utilizzato per lo studio dei reservoir geotermici e di idrocarburi, in ambito di stoccaggio di CO_2, di rifiuti e scorie nucleari, di valutazioni di impatto ambientale e in generale nella modellazione del flusso e trasporto all'interno di mezzi variabilmente saturi.

L'architettura di TOUGH2 è di tipo modulare: è costituito da un modulo principale che descrive i fenomeni di flusso e trasporto che può interfacciarsi con altri moduli in cui sono raccolte le differenti proprietà dei fluidi. La natura e le proprietà delle specifiche combinazioni di fluidi vengono implementate all'interno delle equazioni che governano il moto sotto forma di parametri termodinamici, come densità, viscosità o entalpia del fluido. Tali parametri a loro volta sono descritti tramite l'appropriato modulo "EOS", acronimo per Equation of State.

Il codice originale di TOUGH2 rilasciato nel 1991 conteneva cinque diversi moduli "EOS", denominati "EOS1" – "EOS5". Tali moduli sono stati progettati per le applicazioni geotermiche e lo smaltimento di rifiuti radioattivi e sono stati migliorati nella versione 2.0 del programma rilasciata nel 1999. Quest'ultima contiene una serie di nuovi moduli "EOS", fornisce una descrizione del flusso e del processo più completa e flessibile e aggiunge nuove funzioni fra le quali migliori risolutori di equazioni differenziali e strumenti per l'output grafico dei risultati. In Tabella 2.1 vengono riportati i moduli "EOS" disponibili nella release 2.0 di TOUGH2 e i set di componenti che è possibile modellizzare con essi.

La modellazione del flusso di fluidi e calore avviene mediante la risoluzione di equazioni di bilancio di massa ed energia. Il fenomeno di avvezione dei fluidi è descritto mediante l'estensione multifase della legge di Darcy; viene inoltre preso in considerazione il trasporto di massa diffuso che interessa tutte le fasi. Il passaggio di calore avviene per conduzione e convezione e in quest'ultimo si tiene conto degli effetti dovuti al calore latente e sensibile.

Modulo	Set di componenti
EOS1*	Acqua, acqua con tracciante
EOS2	Acqua, CO2
EOS3*	Acqua, aria
EOS4	Acqua, aria, con pressione di vapore decrescente
EOS5*	Acqua, idrogeno
EOS7*	Acqua, brine, aria
EOS7R*	Acqua, brine, aria, radionuclidi genitore-figlio
EOS8*	Acqua, "dead" oil, gas non condensabile
EOS9	Flusso isotermo con saturazioni variabili rispondente all'equazione di Richards
EWASG*	Acqua, sale (NaCl), gas non condensabile (include i fenomeni di precipitazione e dissoluzione e cambiamenti di porosità e permeabilità; eventuali trattamenti per gli effetti della diminuzione della pressione di vapore)

Tabella 2.1 – Equation Of State di TOUGH2, i moduli segnati con l'asterisco possono modellizzare sistemi a temperatura costante.

Il processo di simulazione numerica richiede la discretizzazione delle variabili continue nello spazio e nel tempo.

In tutti i codici della famiglia TOUGH la discretizzazione spaziale avviene direttamente dalla forma integrale delle equazioni di conservazione, senza prima convertirle in equazioni differenziali alle derivate parziali. TOUGH2 implementa infatti il metodo IFDM, Integral Finite Difference Method, che evita ogni riferimento a un sistema di coordinate globale essendo così applicabile a discretizzazione regolari o irregolari in una, due o tre dimensioni.

La discretizzazione temporale è completamente implicita e viene eseguita con il metodo alle differenze finite del primo ordine.

TOUGH2 adotta quindi un approccio cell-based, la discretizzazione spaziale infatti porta alla creazione di una griglia composta da elementi di volume, o celle, a cui sono

associati in fase di soluzione i valori delle variabili fluido-termodinamiche, tra cui i valori delle velocità dei fluidi.

2.3 VORO2MESH

VORO2MESH [5][6] è un programma che realizza la discretizzazione spaziale di un dominio convesso mediante la tassellatura di Voronoi. È sviluppato in C++ e fa uso della libreria Voro++ [33].

I dati di input per VORO2MESH possono essere di due tipi, direttamente un set di punti (*seed*) che costituiscono i nodi della tassellatura, oppure un set di superfici geologiche da cui poi vengono in automatico individuati i nodi. Il software produce direttamente i file di input specifici per TOUGH2 inerenti la discretizzazione spaziale del dominio e file con informazioni statistiche sugli elementi.

I dati in output al programma sono diversi e comprendono, fra gli altri, area, volume e coordinate dei vertici degli elementi di Voronoi in cui il dominio è stato discretizzato.

Capitolo III

Creazione linee di flusso - Nozioni teoriche

Nel seguente Capitolo sono raccolte le nozioni teoriche direttamente utilizzate per la creazione delle linee di flusso. Il Capitolo si articola in quattro parti: nel §3.1 vengono descritte le operazioni principali operate dal codice; l'intento è quello di fornire una "mappa" sul quale il lettore può orientarsi per collegare i concetti teorici approfonditi singolarmente nei §§3.2, 3.3 e 3.4.

3.1 La strategia applicata

L'ipotesi principale sotto cui di sviluppa il codice è quella di moto permanente del fluido all'interno del mezzo poroso: tale ipotesi permette di eliminare la distinzione fra traiettorie e linee di flusso [10].

Il metodo di calcolo [23] implementato nel codice si articola in due fasi: l'acquisizione del campo di velocità del dominio discretizzato e il calcolo della velocità nel punto corrispondente alle coordinate occupate dalla particella fluida considerata. Le modalità di gestione dei dati di input rispecchiano questa architettura; è infatti possibile fornire i risultati della simulazione cell-based, con cui poi il codice calcolerà il campo di velocità del dominio, oppure direttamente il campo di velocità. In entrambe le modalità, saranno richiesti in input le coordinate di partenza della traiettoria, in questo modo si individua una specifica particella fluida di cui si seguirà la traiettoria.

Il primo modo si applica solo se è necessario calcolare il campo di velocità, e in questo caso il codice è legato al formato del simulatore cell-based. Nessuna limitazione è posta

sulla forma degli elementi e il codice può quindi operare su griglie strutturate e non strutturate; la discretizzazione del dominio proveniente dalla simulazione ottenuta con TOUGH2 è però realizzata mediante tassellatura di Voronoi con velocità riferentesi alle facce dei blocchi. In questa fase, dato un blocco della discretizzazione di Voronoi, gli si attribuisce un singolo valore di velocità calcolato a partire dalle velocità del fluido note alle interfacce dell'elemento. Questo singolo valore di velocità è poi associato al nodo del blocco. Ripetendo questi passaggi per ognuno degli elementi di volume si ottiene una griglia di nodi a cui sono associati singoli valori di velocità: è così definito il campo delle velocità.

Se invece il campo di velocità è fornito come dato di input questa prima fase non è necessaria e cade quindi anche il vincolo imposto sul tipo di tassellatura usata per la discretizzazione del dominio.

Nella seconda fase il codice segue quanto proposto da Painter ed è eseguita qualsiasi sia la modalità di sottoposizione dei dai di input. In sintesi, in questa fase viene calcolata la velocità del fluido nel punto occupato dalla particella fluida considerata. Data la griglia di nodi forniti dal campo di velocità, sulla base di questa si esegue una seconda discretizzazione del dominio mediante tassellatura di Delaunay. La particella si troverà in ogni istante all'interno di un elemento della tassellatura di Delaunay; la sua velocità viene poi calcolata tramite interpolazione baricentrica fra i valori di velocità associati ai vertici dell'elemento di Delaunay che la contiene.

Trovata la velocità nel punto e moltiplicata per un intervallo temporale predeterminato si ottiene uno spostamento, sommando quest'ultimo alla posizione iniziale avremo la nuova posizione della particella. La posizione trovata sarà il punto di partenza per una ulteriore iterazione.

Tutte le posizioni occupate dalla particella considerata vengono poi raccolte in forma vettoriale e rappresentate graficamente.

Viene infine calcolato il tempo di volo associato ad ogni punto costituente la traiettoria: sono così calcolate tutte le informazioni necessarie per l'applicazione dei metodi streamline-based.

3.2 La cinematica dei fluidi

Fissata una terna cartesiana di riferimento con assi x, y, z le componenti del vettore velocità $\boldsymbol{v}$ in un generico punto di coordinate (x, y, z) assumono la forma [10]:

$$v_x = \frac{dx}{dt}, \tag{3.1}$$

$$v_y = \frac{dy}{dt}, \tag{3.2}$$

$$v_z = \frac{dz}{dt}. \tag{3.3}$$

Il campo vettoriale delle velocità di un fluido è quindi univocamente individuato dalla funzione

$$\boldsymbol{v} = \boldsymbol{v}(x, y, z, t) \tag{3.4}$$

o dalla terna di funzioni scalari equivalente

$$v_x = v_x(x, y, z, t), \tag{3.5}$$

$$v_y = v_y(x, y, z, t), \tag{3.6}$$

$$v_z = v_z(x, y, z, t). \tag{3.7}$$

Fissato un determinato istante t tali funzioni descrivono la posizione di tutti i punti dello spazio occupati dal fluido; al contrario, fissando una terna di valori x, y, z esse descrivono quanto accade in quel determinato punto nello spazio al trascorrere del tempo [10].

3.2.1 Elementi caratteristici del moto: traiettorie, linee di flusso e linee di emissione

Nella visualizzazione del moto nel suo insieme ci si avvale, come anticipato, di tre famiglie di linee che pongono in evidenza aspetti diversi del movimento: le *traiettorie*, le *linee di flusso* e le *linee di emissione* [10][19].

Le *traiettorie* sono le linee percorse al trascorrere del tempo dai diversi elementi fluidi. Il riferimento alla traiettoria è efficace, in genere, solo se avviene in intervalli temporali limitati che non permettono la diffusione di tale elemento fluido in tutta la massa in movimento. Al procedere del tempo infatti la linea della traiettoria tende a perdere chiarezza di lettura: traiettorie di particelle diverse possono intersecarsi oppure la medesima particella può ripassare in tempi diversi nella stessa posizione.

Il complesso delle traiettorie è definito dalla seguente terna di equazioni differenziali:

$$dx = v_x(x, y, z, t)\, dt\,, \tag{3.8}$$

$$dy = v_y(x, y, z, t)\, dt\,, \tag{3.9}$$

$$dz = v_z(x, y, z, t)\, dt\,. \tag{3.10}$$

Per ricavare invece la traiettoria specifica della singola particella basterà imporre come condizione iniziale la sua posizione (x_0, y_0, z_0) al tempo t_0.

La seconda famiglia di linee utili alla descrizione del moto è quella delle *linee di flusso* (o *linee di corrente*). Ciascuna di esse, supposto noto il vettore velocità al tempo generico t in ogni punto del campo di moto, è la curva tangente, in ciascuno dei suoi punti, al vettore velocità in quel punto. Per ogni punto del campo di moto passa una sola linea di flusso fatta eccezione per i punti singolari, ne risulta che la rappresentazione del moto mediante le linee di flusso è più efficace rispetto all'uso delle traiettorie. Le linee di flusso infatti sono, sempre facendo riferimento a un dato momento, distinte fra loro e si intersecano solo nei punti in cui la particella fluida ristagna per più istanti, ovvero in quei punti dove momentaneamente il vettore velocità $\boldsymbol{v}$ è nullo. Dal punto di vista analitico le linee di flusso sono espresse dalle seguenti relazioni differenziali

$$\frac{dx}{v_x(x,y,z,\ t_0)} = \frac{dy}{v_y(x,y,z,\ t_0)} = \frac{dz}{v_z(x,y,z,\ t_0)} \tag{3.11}$$

dove t_0 va considerato come parametro.

Le tre componenti del vettore velocità v_x, v_y, v_z variano, in generale, da istante a istante, ne consegue quindi che le linee di flusso cambieranno forma al procedere del tempo.

Si definisce infine per completezza la terza famiglia di linee, quella delle *linee di emissione* (o *linee di fumo*). Preso in considerazione un generico punto P e l'insieme di tutte le particelle fluide passati per P, dicesi linea di emissione la curva che collega tutte queste particelle a un generico istante t. Come per le traiettorie l'utilizzo delle linee di emissione è efficace solo se limitato a un breve intervallo temporale; esse infatti perdono consistenza e utilità man mano che le varie particelle si disperdono nella massa fluida perdendo di coerenza.

Una importante considerazione ai fini di questa trattazione è la seguente. Le velocità delle particelle fluide poste su una linea di corrente sono dirette lunga la linea stessa e le loro traiettorie sono quindi tangenti a essa. Se le direzioni delle velocità non cambiano nel tempo le linee di corrente stesse non mutano; inoltre, tutti i punti che passano per un punto fissato hanno una traiettoria con tangente locale sempre identica e coincidente alla tangente locale della linea di corrente: ne consegue quindi che linee di corrente e traiettorie si sovrappongono. Per quanto riguarda invece le linee si emissione a cui tali elementi appartengono, essa è a sua volta tangente alla linea di corrente di partenza. Ne risulta quindi che se le direzioni delle velocità non cambiano al trascorrere del tempo, ovvero siamo nelle condizioni di moto stazionario, le linee di emissione si sovrappongono alle traiettorie che, a loro volta, coincidono con le linee di flusso [10][19].

3.2.2 Tipi di moto: moto stazionario

L'intero caso di studio, come anticipato, farà riferimento a una situazione di moto stazionario (o permanente). Tale scelta è stata presa alla luce di due considerazioni: la

prima è che il moto permanente descrive con buona approssimazione il lento moto di un fluido di un mezzo poroso anche per intervalli temporali molto lunghi. La seconda considerazione è di tipo operativo, in caso di moto permanente infatti scompare la distinzione fra le traiettorie e le linee di flusso che altrimenti avrebbero tracciati diversi.

Dicesi *moto stazionario* o *permanente* [10] quello in cui le variabili cinematiche non dipendono dal tempo; nella fattispecie le tre componenti del vettore velocità v_x, v_y, v_z sono funzione delle sole coordinate spaziali x, y, z. È interessante notare come un moto possa essere permanente per un determinato sistema di riferimento e contemporaneamente non esserlo per un altro. La parametrizzazione del fattore tempo porta già intuitivamente a una semplificazione del problema, oltre a quella già anticipata riguardante le famiglie di linee caratterizzanti il moto. Un moto non permanente è invece definito come *vario*.

Altra tipologia è invece il *moto uniforme*, che può essere visto come estensione del moto permanente in quanto in esso la velocità, oltre a non essere più dipendente dal tempo, perde anche la dipendenza dalla posizione del punto nel campo di moto considerato. Il moto uniforme assume particolare importanza nello studio del movimento di fluido all'interno di tubazioni, mentre le leggi che lo caratterizzano non vengono impiegate per descrivere il movimento all'interno di un mezzo poroso. Si citano infine per completezza i casi di *moto permanente* o *uniforme in media*.

Verrà inoltre fatto uso del concetto di *moto piano*, i movimenti appartenenti a questa tipologia sono caratterizzati da un vettore velocità ovunque parallelo a uno specifico piano P. Facendo riferimento alla terna di assi cartesiana, se z è l'asse perpendicolare al suddetto piano P la componente v_z della velocità sarà nulla in ogni punto del campo di moto mentre le altre due componenti v_x, v_y saranno indipendenti da z, le relazioni che descrivono tale moto sono quindi:

$$v_x = v_x(x, y, t), \tag{3.12}$$

$$v_y = v_y(x, y, t), \tag{3.13}$$

$$v_z = 0\,. \tag{3.14}$$

Naturalmente quella descritta da tali relazioni è una natura semplificata del fenomeno, ciononostante essa può essere molto utile per la comprensione degli elementi essenziali del moto [10].

3.3 La discretizzazione del dominio

È già stato accennato come la strategia implementata nel codice comporti la discretizzazione del dominio. In questo Paragrafo vengono presentate le due diverse tassellature usate nel lavoro di tesi e le definizioni di discretizzazioni strutturate e non strutturate.

3.3.1 Discretizzazione strutturata e non strutturata

È definita discretizzazione strutturata una opportuna suddivisione del dominio i cui i blocchi possono essere univocamente identificati mediante una terna di indici i, j, k [16][34]: se al contrario non è possibile definire una relazione biunivoca fra terna di indici e specifico blocco si parla di discretizzazione non strutturata.

Figure tipiche degli elementi in una discretizzazione strutturata sono in 2D i quadrilateri e in 3D i prismi; in una discretizzazione non strutturata invece si scelgono generalmente gli elementi triangolari o tetraedrici [8].

In linea generale è difficile generare delle griglie strutturate di buona qualità per domini complessi; esse inoltre mal si prestano a raffittimenti locali dei nodi. In presenza di mezzi porosi con discontinuità, come possono essere faglie, barriere o pozzi di emungimento e iniezione, si preferisce l'utilizzo di griglie non strutturate che, al contrario di quelle strutturate, sono caratterizzate da un'alta flessibilità geometrica e permettono, dove necessario, addensamenti dei nodi.

3.3.2 La tassellatura di Voronoi

Verranno ora fornite la definizione e alcune proprietà della tassellatura di Voronoi [2]; per semplicità si farà riferimento al caso piano, l'estensione a quello 3D è immediata.

Sia definito nel piano un insieme S di n punti chiamati *nodi*. Per due diversi nodi $p, q \in S$, la *regione di dominanza* di p su q è definita come la porzione di piano vicina a p almeno quanto lo è a q. Tale condizione può espressa con la relazione

$$dom(p,q) = \{x \in R^2 \mid \partial(x,p) \leq \partial(x,q)\} \tag{3.15}$$

dove ∂ denota la distanza euclidea. La $dom(p,q)$ è rappresentata da un semipiano limitato dalla bisettrice del segmento congiungente p e q. Tale bisettrice separa tutti i punti del piano più vicini a p da quelli più vicini a q ed è definita come *separatrice*. La *regione* di un nodo $p \in S$ è la porzione di piano che giace completamente nella regione di dominanza di p sugli altri nodi di S, ovvero

$$reg(p) = \bigcap_{q \in S - \{p\}} dom(p,q)\,. \tag{3.16}$$

Poiché le regioni sono individuate dall'intersezione di $n-1$ semipiani, esse sono poligoni convessi. I limiti di tali regioni possono essere costituiti al massimo da $n-1$ spigoli e vertici; ogni punto su uno spigolo è equidistante da esattamente due nodi e ogni vertice invece da almeno tre di essi. La partizione poligonale del piano che ne deriva è definita come tassellatura di Voronoi, $V(S)$, per l'insieme finito di punti S [2].

Per un insieme finito di punti in 3D il discorso è analogo, a ogni nodo è associato un elemento di Voronoi contenente tutti i punti più vicini a quel nodo rispetto agli altri nodi; tutti questi elementi costituiscono la tassellatura di Voronoi per quello specifico insieme di nodi.

3.3.3 La tassellatura di Delaunay

La tassellatura di Delaunay [31] si fonda sull'omonimo criterio che in 2D è anche noto come criterio del circumcerchio vuoto. La tassellatura di Delaunay per un set di nodi in uno spazio 2D assicura che il circumcerchio associato ad ogni triangolo non contenga al suo interno nessun altro nodo del set. Un'altra caratteristica è che la tassellatura di Delaunay connette le terne di nodi più strettamente vicini fra di loro.

In Figura 3.1 si può notare che i circumcerchi associati ai triangoli T1 e T2 non contengono nessun nodo al suo interno: si tratta quindi di una tassellatura di Delaunay. In Figura 3.2, al contrario, i circumcerchi di T1 e T2 contengono rispettivamente i vertici V1 e V4, non siamo quindi in presenza di una tassellatura di Delaunay.

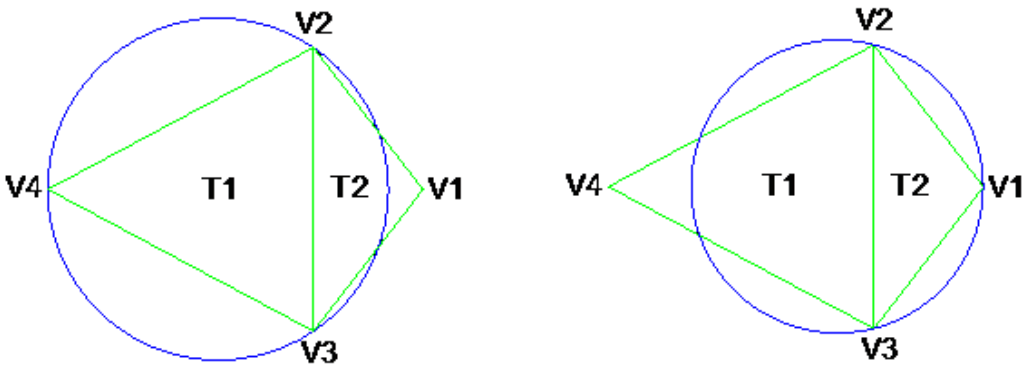

Figura 3.1 - Esempio di tassellatura di Delaunay.

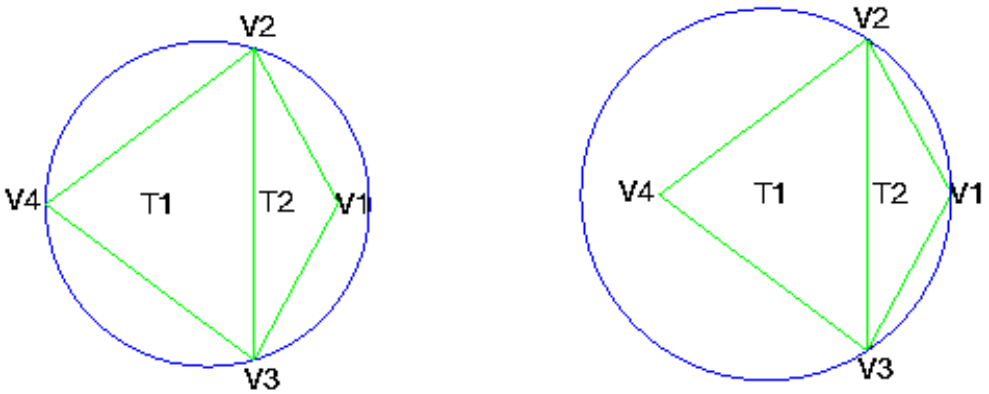

Figura 3.2 - Esempio di tassellatura che non rispetta il criterio di Delaunay.

Nel caso in cui la tassellatura sia eseguita su un set di punti degeneri essa non è unica; in due dimensioni ad esempio, la degenericità sorge nel momento in cui quattro punti giacciono sulla stessa circonferenza, come accade in Figura 3.3.

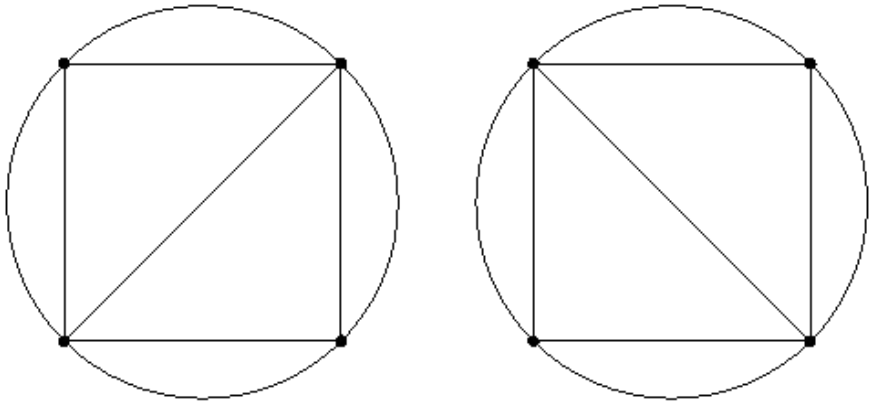

Figura 3.3 - Triangolazioni di Delaunay non univoche.

Per quanto riguarda invece lo spazio tridimensionale, la tassellatura di Delaunay è composta da tetraedri che soddisfano il criterio della circumsfera vuota. In Figura 3.4 è mostrata una semplice tassellatura di Delaunay composta da due tetraedri; è mostrata la circumsfera di uno dei due tetraedri per mettere in evidenza il criterio della circumsfera vuota [31].

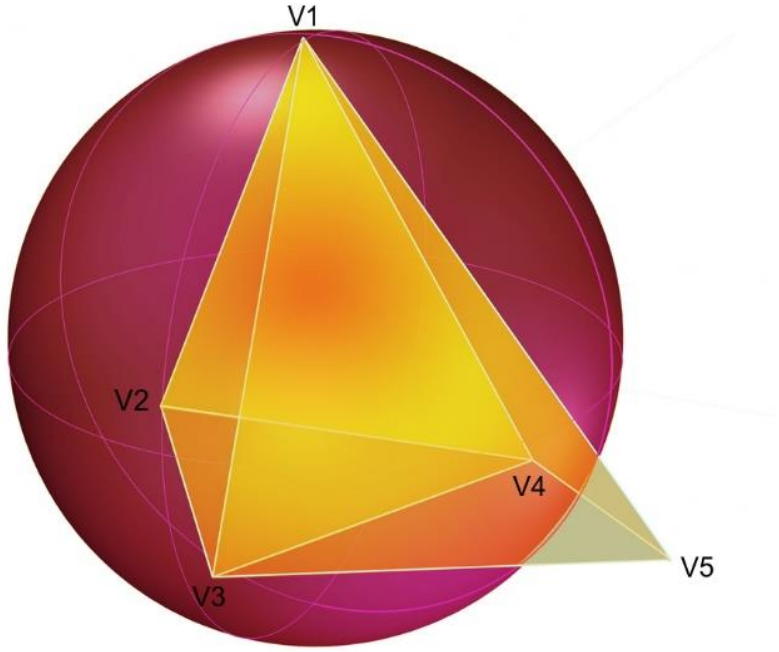

Figura 3.4 - Tassellatura di Delaunay 3D.

3.4 Il metodo di Painter

L'approccio proposto da Painter nell'articolo "Pathline tracing on fully unstructured control-volume grids" [23] utilizza come dati di partenza le portate volumetriche del fluido alle interfacce fra elementi. Questo dato, corredato da quello dell'estensione areale di tale interfaccia, premette il calcolo della velocità. Nel codice invece i dati di partenza adottati sono direttamente quelli di velocità del fluido all'interfaccia resi disponibili dalla simulazione con TOUGH2.

Painter suppone che il dominio di interesse sia già stato suddiviso in elementi triangolari (in 2D) o tetraedrici (in 3D) e considera le bisettrici perpendicolari ai lati o spigoli di tali elementi: l'intersezione di tali bisettrici crea un'altra suddivisione del dominio in volumi di controllo V_i. Si vengono a creare così due suddivisioni del dominio: la prima, che per comodità chiameremo A, è formata da elementi triangolari o tetraedrici, la seconda, B, formata dai volumi di controllo V_i. Sulla suddivisione B vige la restrizione di costruzione mediante bisettrici perpendicolari, ma nessuna circa la forma dei volumi di controllo. La suddivisione B risulta una tassellatura di Voronoi se A è fatta mediante tassellatura di Delaunay. L'ipotetica discretizzazione di un dominio 2D è riportata in Figura 3.5.

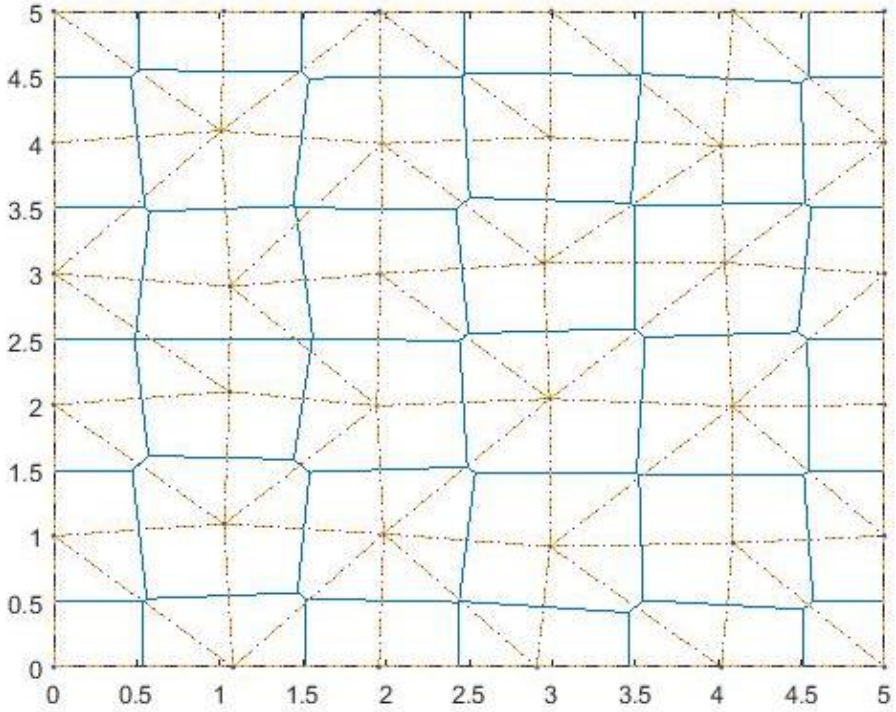

Figura 3.5 – In linea tratteggiata rossa la tassellatura di Delaunay corrispondente alla partitura A, in linea solida blu la tassellatura di Voronoi, corrispondente alla partitura B.

Come già accennato l'approccio proposto da Painter si articola in due fasi.

Nella prima fase viene calcolato un valore rappresentativo della velocità $\boldsymbol{v}_i$ attribuibile al blocco a partire dalle componenti normali alle facce del blocco v_{ij} note. Tali valori sono associati al nodo, eseguendo l'operazione per tutti i blocchi è definito il campo di velocità.

Nella seconda fase a partire dal campo di velocità individuato nella prima fase viene calcolato il valore della velocità in ogni punto del dominio.

3.4.1 Ricostruzione delle velocità al centro delle celle

Temporaneamente assumiamo che in ogni cella $\boldsymbol{v}$ sia costante, tale limite sarà valido solo nella prima fase mentre cadrà nella seconda.

Consideriamo la generica cella i confinante con quattro celle j, k, p, q. Possiamo quindi esprimere la velocità del fluido attraverso la faccia comune alle celle i e j come uno scalare v_{ij}. Il versore che rappresenta la faccia sarà invece identificato con $\boldsymbol{n}_{ij}$.

Queste entità possono essere individuate per ogni faccia della cella i, raccogliendo quindi i versori $\boldsymbol{n}$ nella matrice $\boldsymbol{G}_i$e gli scalari v nel vettore $\boldsymbol{\gamma}_i$ ne risulta:

$$\boldsymbol{G}_i = \begin{bmatrix} \boldsymbol{n}_{ij} \\ \boldsymbol{n}_{ik} \\ \boldsymbol{n}_{ip} \\ \boldsymbol{n}_{iq} \end{bmatrix} \qquad \boldsymbol{\gamma}_i = \begin{bmatrix} v_{ij} \\ v_{ik} \\ v_{ip} \\ v_{iq} \end{bmatrix}. \tag{3.17)(3.18}$$

È quindi definito il sistema lineare

$$\boldsymbol{G}_i \boldsymbol{v}_i = \boldsymbol{\gamma}_i \tag{3.19}$$

dove $\boldsymbol{G}_i$ ha dimensioni $n_i \times d$, $\boldsymbol{v}_i$, incognita di questo sistema, ha dimensioni $d \times 1$ e infine $\boldsymbol{\gamma}_i$ ha dimensioni $n_i \times 1$ con n_i numero di celle adiacenti a quella in esame i e d è la dimensione dello spazio.

In due dimensioni il volume di controllo ha un minimo di tre lati, mentre in tre dimensioni ha un minimo di quattro facce. Ne consegue che $n_i > d$ e perciò il sistema (3.19) di dimensioni $n_i \times d$ è sovradeterminato. La velocità di Darcy $\boldsymbol{v}_i$ può essere

quindi stimata mediante il metodo dei minimi quadrati facendo ricorso alla pseudo-inversa di Moore-Penrose:

$$\widehat{\boldsymbol{v}}_{\boldsymbol{i}} = (\boldsymbol{G}^t{}_i \boldsymbol{G}_{\boldsymbol{i}})^{-1} \boldsymbol{G}^t{}_i \gamma_i \, . \tag{3.20}$$

Con condizioni al contorno di tipo Dirichlet la relazione (3.20) è valida sia per i nodi interni che per i nodi ai limiti del dominio, con condizioni al contorno di tipo Neumann, come quelle imposte nel codice, è valida solo per i nodi interni. Per i nodi ai confini del dominio bisogna infatti tenere conto dei vincoli di portata imposti, il che porta a un problema di minimi quadrati lineare per il vettore di flusso al contorno $\widehat{\boldsymbol{v}}_{\boldsymbol{i}}{}^{b}$ della cella i-sima definito come

$$\widehat{\boldsymbol{v}}_{\boldsymbol{i}}{}^{b} = \arg\min_{\boldsymbol{q_i}} \|\boldsymbol{G}_i \boldsymbol{v}_i - \gamma_i\| \quad \text{con} \quad \boldsymbol{B}_i \boldsymbol{v}_i = \beta_i \tag{3.21}$$

dove $\boldsymbol{B_i}$ è una matrice $n_i{}^{c} \times d$ e β_i un vettore $n_i{}^{c} \times 1$ con $n_i{}^{c} < d$ numero di lati o facce ai limiti del dominio. La matrice $\boldsymbol{B_i}$ e il vettore β_i sono gli analoghi di $\boldsymbol{G}_i$ e γ_i ma sono scritti solo per il sottoinsieme di facce della cella i-esima che giacciono sul limite del dominio. La soluzione esplicita può essere scritta come

$$\widehat{\boldsymbol{v}}_{\boldsymbol{i}}{}^{b} = \widehat{\boldsymbol{v}}_{\boldsymbol{i}} - (\boldsymbol{G}^t{}_i \boldsymbol{G}_{\boldsymbol{i}})^{-1} \boldsymbol{B}_{\boldsymbol{i}} \left(\boldsymbol{B}_{\boldsymbol{i}}{}^{\boldsymbol{t}} (\boldsymbol{G}^t{}_i \boldsymbol{G}_{\boldsymbol{i}})^{-1} \boldsymbol{B}_{\boldsymbol{i}} \right)^{-\mathbf{1}} (\boldsymbol{B}_{\boldsymbol{i}} \widehat{\boldsymbol{v}}_{\boldsymbol{i}} - \beta_i) \, . \tag{3.22}$$

Nel caso in cui $n_i{}^{c} = d$ le condizioni al contorno determinano da sole le velocità ai nodi. Le equazioni (3.20) e (3.22) possono essere applicate a ogni cella i-esima per ottenere una rappresentazione discretizzata del campo di velocità nel dominio.

3.4.2 Interpolazione delle velocità

L'assunzione fatta nella prima fase di velocità costanti all'interno di ogni cella porta a discontinuità passando da una cella all'altra. Tali discontinuità possono a loro volta portare a errori nei bilanci di massa e sono inadeguate nella fase di calcolo delle linee di

flusso; è quindi necessario eliminarle tramite una operazione di interpolazione. Perciò associamo ogni velocità $\boldsymbol{v_i}$ al rispettivo nodo della discretizzazione A e a partire da tali dati interpoliamo la velocità in ogni punto del dominio.

In 3D definite $\boldsymbol{r_1}, \boldsymbol{r_2}, \boldsymbol{r_3}$ e $\boldsymbol{r_4}$ le posizioni dei vertici di un tetraedro nella griglia, la velocità di un punto in posizione $\boldsymbol{r}$ all'interno del tetraedro può essere interpolata come

$$\boldsymbol{v}(\boldsymbol{r}) = \sum_{i=1,4} \lambda_i \boldsymbol{v}_i \tag{3.23}$$

$$\begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \end{pmatrix} = \boldsymbol{T}^{-1}(\boldsymbol{r} - \boldsymbol{r}_4) \tag{3.24}$$

con $\sum_{i=1,4} \lambda_i = 1$ e $\mathbf{T}$ matrice 3×3 le cui colonne sono costituite dalle differenze $\boldsymbol{r} - \boldsymbol{r_1}, \boldsymbol{r} - \boldsymbol{r_2}$ e $\boldsymbol{r} - \boldsymbol{r_3}$ [23].

Il caso 2D è analogo, definite $\boldsymbol{r_1}, \boldsymbol{r_2}$ e $\boldsymbol{r_3}$ le posizioni dei vertici di uno dei triangoli costituenti la tassellatura di Delaunay, la velocità di un punto in posizione $\boldsymbol{r}$ all'interno del triangolo può essere interpolata come

$$\boldsymbol{v}(\boldsymbol{r}) = \sum_{i=1,3} \lambda_i \boldsymbol{v}_i \tag{3.25}$$

$$\begin{pmatrix} \lambda_1 \\ \lambda_2 \end{pmatrix} = \boldsymbol{T}^{-1}(\boldsymbol{r} - \boldsymbol{r}_3) \tag{3.26}$$

con $\sum_{i=1,3} \lambda_i = 1$ e $\mathbf{T}$ matrice 2×2 le cui colonne sono costituite dalle differenze $\boldsymbol{r} - \boldsymbol{r_1}$ e $\boldsymbol{r} - \boldsymbol{r_2}$.

Capitolo IV

Il Codice MatLab

Si è finora parlato del codice MatLab come un file unico, in realtà lo strumento di analisi elaborato si articola in due file eseguibili dedicati per simulazioni in due e tre dimensioni chiamati, rispettivamente, P2D e P3D. Sia P2D che P3D sono inoltre corredati da sei file MatLab esterni al codice (denominati *function*) necessari per importare in ambiente MatLab i dati necessari al codice stesso.

Il seguente Capitolo viene suddiviso in cinque Paragrafi.

Nel §4.1 vengono presentati i dati di input necessari al codice e i file di TOUGH2 e VORO2MESH che li contengono. Alcuni di questi file necessitano di una fase di preprocessing eseguita dall'utente che sarà anch'essa descritta in dettaglio.

Il §4.2 è dedicato alle function esterne, queste non sono riportate per intero all'interno del Capitolo ma se ne illustra lo scopo e la struttura.

Nel §4.3 viene riportato per intero il codice nella sua versione 3D, verranno singolarmente descritte le cinque *Section* che lo compongono e le operazioni in esse eseguite.

Nel §4.4 verrà brevemente introdotto P2D e infine nel §4.5 saranno presentati gli output del codice.

Sia P2D che P3D, completi di relative funzioni, sono riportati per intero in Appendice.

4.1 I file di input

È possibile fornire al codice i dati di input in due modalità a seconda che il campo di velocità sia già stato calcolato o meno.

Nel caso in cui il campo di velocità dovesse essere calcolato i file di input necessari al funzionamento del codice sono quattro:

- seed.txt;
- elements.txt;
- faces.txt;
- velocities.txt.

Se invece il campo di velocità è già determinato sono necessari due soli file:

- seed.txt;
- velocityfield.txt.

4.1.1 seed.txt

Nel file seed.txt, costruito direttamente dall'utente del codice, sono specificati i punti iniziali delle traiettorie che si vogliono studiare. Tali dati vengono importati in ambiente MatLab nella matrice numerica chiamata "seed".

Per ogni punto iniziale è riservata una riga di seed.txt divisa in tre campi, uno per la coordinata x, uno per la coordinata y e uno per la coordinata z, ognuno costituito da 10 caratteri.

```
----X----*----Y----*----Z----*
  50         100              -10
```

4.1.2 elements.txt

Nel caso in cui il campo di velocità non è definito il codice necessita del file elements.txt, questo contiene informazioni sulla posizione dei vari blocchi di Voronoi nel sistema di riferimento assoluto. Le variabili importate in MatLab da questo file sono le seguenti.

- ELEMENTS: vettore composto dagli identificativi degli elementi.
- X, Y, Z: vettori contenenti le coordinate cartesiane del nodo a cui è associato l'elemento della griglia.

Nel simulatore utilizzato, questi dati possono essere ricavati dal blocco relativo alla keyword ELEME che fa parte degli input di TOUGH2.

L'input di TOUGH2 è formato da uno o più file ASCII con non più di 80 caratteri per riga. L'utilizzatore di TOUGH2 può fornire i dati geometrici al simulatore nelle tre modalità illustrate in Figura 4.1, nel file INPUT mediante i blocchi ELEME e CONNE, mediante il blocco MESHM o fornendo direttamente il file MESH [25].

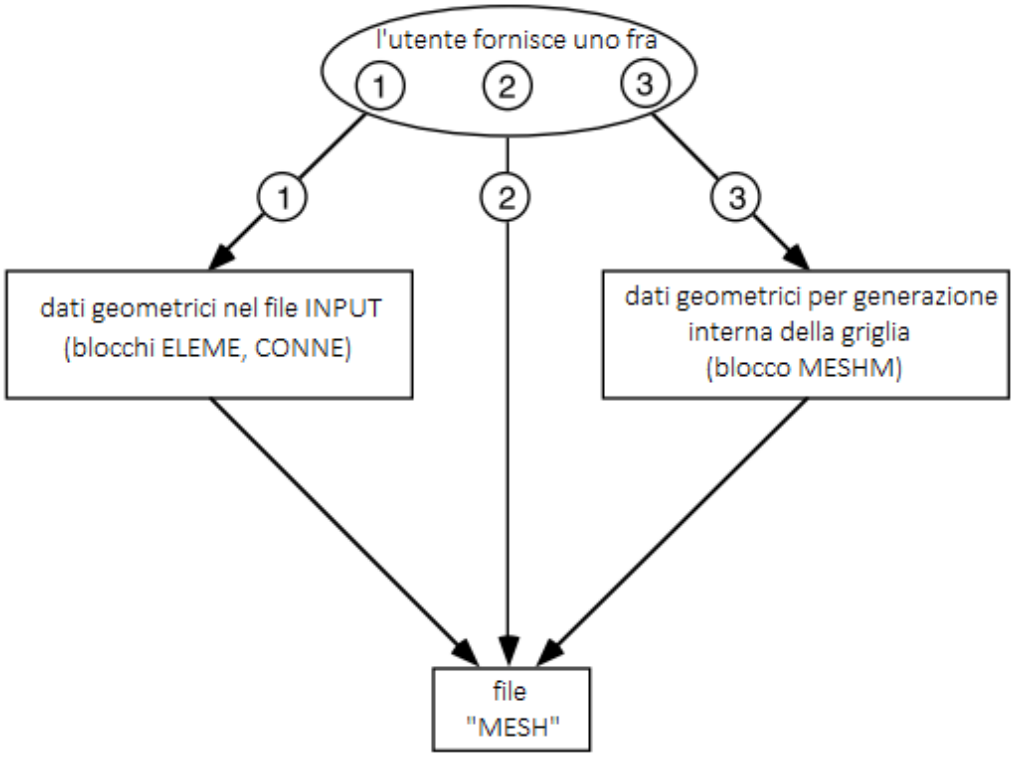

Figura 4.1 – Opzioni dell'utente di TOUGH2 per fornire i dati geometrici al simulatore.

VORO2MESH fornisce un file MESH pronto all'uso con TOUGH2 ed è proprio in questo file che si può reperire il blocco ELEME necessario. Per la creazione del file elements.txt è sufficiente copiare l'intero blocco ELEME e incollarlo su un nuovo file di testo denominato appunto elements.txt. Non è necessario "pulire" il blocco ELEME da righe vuote o con elementi diversi dai dati (ad esempio, unità di misura), nella fase di importazione dei dati il codice riconosce e ignora questi record.

L'organizzazione degli 80 caratteri della riga del blocco ELEME è quella riportata in Figura 4.2.

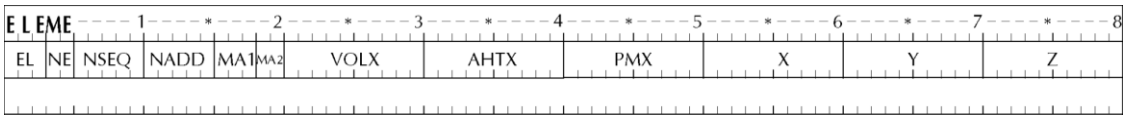

Figura 4.2 - Organizzazione dei caratteri nel blocco ELEME.

Nei record di ELEME ogni componente di ELEMENTS è costituito da 5 caratteri identificati in Figura 4.2 dai campi EL e NE. EL è costituito da tre caratteri arbitrariamente alfanumerici, NE invece da due caratteri necessariamente numerici. Le coordinate del nodo sono invece individuate nei campi 6, 7 e 8, rispettivamente X, Y, e Z [29].

4.1.3 faces.txt

Sempre considerando il caso in cui il campo di velocità debba essere calcolato dal codice, un altro file necessario è faces.txt, contenente informazioni sulle facce degli elementi di volume. Le variabili create a partire da questo file sono le seguenti:

- n_faces: numero di facce totali dell'elemento, comprensivo di quelle rivolte verso l'interno del dominio e quelle rivolte verso l'esterno di esso;
- VF: matrice contenente i versori normali alle sole facce di bordo.

In ambito TOUGH2 questi dati sono contenute in tough2viewer.dat che è un file di output di VORO2MESH. Questo file contiene una riga per ogni elemento della tassellatura di Voronoi, le informazioni fornite per ogni elemento sono riportate in Tabella 4.1 [5].

Per essere correttamente utilizzato dal codice il file tough2viewer.dat deve essere rinominato come faces.txt.

id	Nome dell'elemento, corrispondente all'identificativo fornito da TOUGH2.
x, y, z	Coordinate del nodo a cui è associato l'elemento.

n_vertex	Numero di vertici.
[n_vertex(x,y,z)]	Una per vertice, fra parentesi sono indicate le coordinate dei vertici nel sistema di riferimento relativo avente l'origine nel nodo.
n_faces	Numero di facce.
[n_faces(i_vertex, i_vertex+1, …)]	Una per faccia, fra parentesi sono indicati i vertici che individuano ogni singola faccia.
[n_faces(vx,vy,vz)]	Una per faccia, fra parentesi sono indicate le componenti dei versori normali alla faccia.

Tabella 4.1 – Schema riassuntivo delle informazioni fornite in tough2viewer.dat da VORO2MESH.

4.1.4 velocities.txt

Il terzo file, velocities.txt, contiene informazioni sulle connessioni fra elementi. Le variabili create in MatLab sulla base dei dati contenuti in questo file sono:

- ELEM1: nome del primo elemento della connessione.
- ELEM2: nome del secondo elemento della connessione.
- VELLIQ: velocità della fase liquida che attraversa l'interfaccia fra i due elementi, il valore è positivo se il fluido passa da ELEM2 a ELEM1.

Tali informazioni possono essere importate dal file di output di TOUGH2 la cui intestazione è riportata in Figura 4.3.

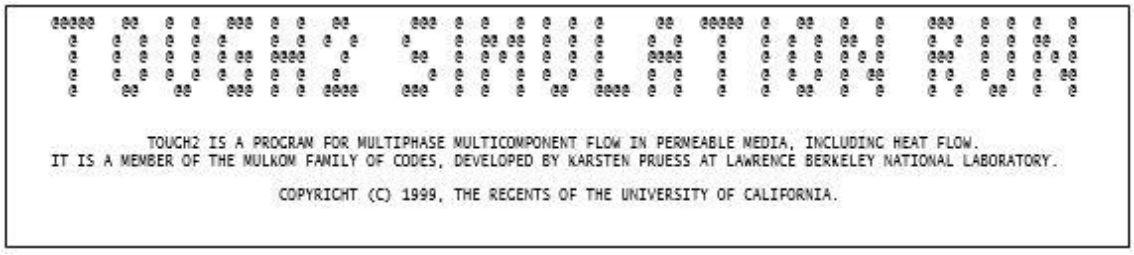

Figura 4.3 – Intestazione del file di output di TOUGH2.

TOUGH2 può produrre una grande varietà di stampe di output, la maggior parte delle quali gestibili dall'utente [25]. Il blocco di interesse per il codice è quello mostrato in Figura 4.4 contente in particolare il dato VEL(LIQ.).

```
          TOUGH2 Analysis

                                                ITER =    3  -  TIME =0.110449E+11
 ELEM1  ELEM2  INDEX  FLOH                          FLO(LIQ.)     VEL(GAS)     VEL(LIQ.)
                      (W)                             (kg/s)       (m/s)        (m/s)

    0 1    059    1 -.249757E+04     ...     -.500717E-02 0.00000E+00 -.19948E-06
    0 1    058    2 -.249044E+04     ...     -.499284E-02 0.00000E+00 -.19891E-06
    0 2    236    3 0.168846E+01     ...     0.225338E-04 0.00000E+00 0.44744E-09
    0 2    239    4 -.145998E+01     ...     -.225313E-04 0.00000E+00 -.46937E-09
    0 3    210    5 0.315496E+03     ...     0.499776E-02 0.00000E+00 0.19848E-06
    0 3    211    6 0.315779E+03     ...     0.500225E-02 0.00000E+00 0.19865E-06
```

Figura 4.4 – Blocco di dati utili al codice presenti nel file di output di TOUGH2.

L'utente di TOUGH2 può impostare l'esecuzione di stampe del risultato della simulazione a determinati step temporali, individuato quello di interesse è necessario riportarne i dati su un file di testo che andrà rinominato velocities.txt. Per farlo è necessario copiare le colonne ELEM1, ELEM2 e VEL(LIQ.) all'interno del nuovo file velocities.txt facendo attenzione a rispettare la seguente suddivisione dei campi: i primi 5 caratteri della riga sono dedicati a ELEM1, i caratteri dal 6° al 10° sono invece dedicati a ELEM2, a VEL(LIQ.) sono destinati tutti i caratteri necessari dall'11° in poi.

```
12345123450123456789
ELEM1ELEM2 VEL(LIQ.)
```

Come per il blocco ELEME non è necessario "pulire" il blocco di output di TOUGH2 da righe vuote o unità di misura; il file velocities.txt si deve presentare come indicato in Figura 4.5.

```
          TOUGH2 Analysis

ELEM1ELEM2   VEL(LIQ.)
                (m/s)

0 1    059  -.19948E-06
0 1    058  -.19891E-06
0 2    236  0.44744E-09
0 2    239  -.46937E-09
0 3    210  0.19848E-06
0 3    211  0.19865E-06
```

Figure 4.5 – Come si deve presentare il file velocities.txt. ELEM1 deve essere completamente contenuto all'interno dei primi cinque caratteri della riga ed ELEM2 all'interno dei secondi cinque. Nessun vincolo è invece posto sul numero di caratteri in cui è contenuto VEL(LIQ.).

4.1.5 velocityfield.txt

Come già anticipato se il campo di velocità è già stato calcolato i dati di input possono essere forniti al codice con una diversa modalità. In questo caso è necessario un solo file (oltre a seed.txt) chiamato velocityfield.txt in cui è appunto memorizzato il campo di velocità del dominio discretizzato.

Per campo di velocità si intende una griglia di nodi a cui sono associati i corrispondenti vettori velocità. Il file in oggetto risulta composto da sei colonne in cui sono riportate le tre coordinate spaziali del nodo e le tre componenti del vettore velocità lungo gli assi di riferimento; la disposizione dei dati è quella mostrata in Figura 4.6.

```
X           Y           Z           VX           VY           VZ
1.00000e-03 1.00000e-03 -1.00000e-03 -0.00000e+00 -0.00000e+00 -0.00000e+00
5.00000e+01 1.00000e-03 -1.00000e-03 6.81090e-06 -2.11758e-22 2.11758e-22
1.00000e+02 1.00000e-03 -1.00000e-03 1.74767e-06 -0.00000e+00 -5.29396e-23
1.50000e+02 1.00000e-03 -1.00000e-03 6.91216e-07 -0.00000e+00 -0.00000e+00
2.00000e+02 1.00000e-03 -1.00000e-03 3.52579e-07 6.61744e-24 -0.00000e+00
2.50000e+02 1.00000e-03 -1.00000e-03 2.09193e-07 3.30872e-24 6.61744e-24
```

Figure 4.6– Come si deve presentare il file velocityfield.txt.

Il file deve rispondere a delle specifiche di costruzione necessarie alla corretta importazione: la prima riga è riservata all'intestazione delle colonne mentre la separazione fra i dati all'interno della stessa riga deve essere eseguita mediante uno o più spazi vuoti.

Il file velocityfield.txt è stato finora presentato come file di input in cui è rappresentato il campo di velocità. Se invece il campo di velocità non è già disponibile ma deve essere calcolato a partire dai risultati della simulazione numerica il file velocityfield.txt viene creato dal codice stesso, diventando, in questo modo, file di output. Questo doppio "ruolo" di velocityfield.txt di file di input e output risulta molto utile nel caso in cui si dovessero eseguire diversi Run del codice. Basterebbe infatti calcolare il campo di velocità una singola volta perché questo venga memorizzato nel file velocityfield.txt, nel caso di un ulteriore Run il codice può importare il campo di velocità direttamente senza ricalcolarlo con un notevole risparmio in termini computazionali.

4.2 Le funzioni per importare i dati

MatLab permette di definire funzioni specifiche per il problema trattato, esse assumono stesso rango di quelle built-in e come quest'ultime accettano argomenti in uscita e restituiscono argomenti in uscita.

Il codice è corredato da cinque funzioni il cui scopo è quello di importare i dati necessari dai file presentati nel §4.1, esse sono: *importseed*, *importelements*, *importvelocities*, *importfaces* e *importvelocityfield*. La corrispondenza con i file di input è data dalla seconda parte del nome della funzione, il loro utilizzo è subordinato alla modalità con cui si è scelto di sottoporre i dati di input al codice.

La scelta di funzioni esterne piuttosto che la loro implementazione all'interno del codice permette il facile uso di quest'ultimo con dati provenienti da simulatori/programmi diversi da TOUGH2 e VORO2MESH.

MatLab permette di creare variabili come vettori e matrici importando nel proprio Workspace i dati da file con estensioni delle più disparate, dal .mp3 al, come nel caso di questo elaborato, .txt. Questa operazione può essere eseguita manualmente selezionando il file da importare e le modalità con cui farlo, che ovviamente cambiano in base all'estensione del file da importare. Per i file di testo possono essere definiti, fra gli altri, il numero di colonne da importare, il loro tipo (di testo o numeriche) e la loro larghezza, i separatori di colonna utilizzati (spazi punteggiatura o personalizzati dall'utente) e cosa fare degli elementi non importabili. Queste specifiche di importazione possono essere definite volta per volta che si importano i file oppure MatLab permette di generare una funzione (un file .m) che, a partire da file di input con caratteristiche ricorrenti, crea in uscita sempre le stesse variabili.

Le funzioni per l'importazione dei dati utilizzate in questo codice sono state parzialmente generate adottando questo metodo.

Nel caso in cui il campo di velocità non fosse determinato e se di conseguenza fosse scelta la prima modalità di sottoposizione dei dati al codice, alle funzioni di importazione dati se ne aggiunge una ulteriore denominata *ffaces*. Quest'ultima funzione completa l'operato di importfaces, una descrizione più approfondita di ffaces è data nel §4.3.2.

4.3 P3D

Come già anticipato il codice è diviso in cinque Section, queste sono macroblocchi di istruzioni definiti dall'utente e divisi mediante il simbolo %%. È possibile eseguire il Run di una singola Section, facilitando la correzione del codice e dando organizzazione logica alle operazioni.

4.3.1 Section 1, definizione dei parametri da parte dell'utente

Il codice si apre con un preambolo di tre comandi che servono rispettivamente a pulire tutte le variabili che possono essere state salvate in precedenza, a pulire la Command Window di MatLab e infine per chiudere tutte le finestre dove vengono mostrate le figure ottenute.

```
clear all
clc
close all
```

La Section 1 è quella che maggiormente interessa l'utilizzatore del codice, in essa infatti è possibile definire una serie di parametri utili o necessari per la determinazione delle traiettorie.

```
%%
% ############################ SECTION 1 ##############################
% DEFINITION OF THE PARAMETERS BY THE USER

% Min. displacement
ds_min = 1; % meters

% Max. iterations
n_it_max = 5*10^3;

% Final position
XF = 500;    YF = 500;    ZF = -500; %meters

% Final position tolerances
tolx = 5;    toly = 5;    tolz = 5; %meters
```

Il primo parametro richiesto è lo spostamento minimo che si intende far compiere dalla particella fluida nel corso di una iterazione. Una volta calcolata la velocità della particella si dividerà lo spostamento minimo per la velocità appena trovata ottenendo un intervallo temporale. Moltiplicando nuovamente l'intervallo temporale con le tre componenti della velocità otterremo lo spostamento nelle tre direzioni dello spazio che sommato alla posizione di partenza darà la nuova posizione occupata dalla particella considerata. Un spostamento minimo più ampio influenzerà positivamente la velocità computazionale a discapito della precisione nella determinazione della traiettoria. Valori troppo ampi per lo spostamento potrebbero portare alla fuoriuscita della particella dal dominio alla prima iterazione. Ad ogni modo il codice riporterà un messaggio di errore, basterà in questo caso abbassare il valore inserito.

Il secondo parametro richiesto riguarda il numero di iterazioni massime consentite al codice per la determinazione di una singola traiettoria. Questo valore risulta utile nel caso il processo andasse in loop oppure per interromperlo nel caso non si conoscessero le coordinate del punto finale della traiettoria.

Vengono richieste inoltre le coordinate presunte del punto finale della traiettoria corredate da singole tolleranze per ogni direzione. L'avvicinarsi della posizione della particella al punto finale a meno delle tolleranze diventa un ulteriore criterio di arresto per il processo iterativo. Mediante l'uso delle tolleranze è possibile impostare che la traiettoria si interrompa al raggiungimento di un determinato punto, linea o piano.

Si ritiene opportuno puntualizzare che impostando coordinate e tolleranze per il punto finale della traiettoria si va a specificare un solo criterio di arresto del processo iterativo. Con questo espediente infatti non si verifica il comportamento delle traiettorie nel caso che effettivamente vi fossero nelle posizioni specificate elementi di tipo *sink* (celle per cui non è possibile determinare una direzione di deflusso).

Viene poi chiesto se il campo di velocità è già stato determinato

```
% Velocity field is already defined? 0 for NO, 1 for YES
VField = 1;
```

e vengono infine importati nella matrice seed i punti iniziali delle traiettorie specificati nel file seed.txt.

```
% Pathline seeds import
[seed] = importseed('seed.txt', 1, inf)

% ######################## END OF SECTION 1 ##########################
```

4.3.2 Section 2, determinazione del campo di velocità

L'intera Section 2 è racchiusa in un'unica istruzione condizionale if costituita da due blocchi di codice eseguiti in modo alternativo: il primo blocco viene eseguito nel caso in cui il campo di velocità è già definito e memorizzato nel file velocityfield.txt, se invece è il codice a dover calcolare il campo di velocità è eseguito il secondo blocco di istruzioni.
Il primo blocco è costituito da un solo comando che provvede a importare in ambiente MatLab il campo di velocità calcolato mediante un precedente *Run* del codice o mediante una simulazione di altro tipo.

```
% ############################ SECTION 2 ############################
% DEFINITION OF THE VELOCITY FIELD

if VField == 1; % Velocity field is already defined
   M = importvelocityfield('velocityfield.txt', 2, inf);
```

Se invece è necessario calcolare il campo di velocità si esegue il secondo blocco dell'istruzione if.

```
else % VField == 0;  % Velocity field is NOT defined
```

All'interno di questo blocco di istruzioni la prima operazione compiuta è l'importazione in ambiente MatLab di tutte le informazioni necessarie al codice dai file di testo, elements.txt, velocities.txt e faces.txt: tali file devono essere nella stessa cartella in cui è contenuto P3D.

```
    [ELEMENTS,X,Y,Z] = importelements('elements.txt',1,inf);
    [ELEM1,ELEM2,VELLIQ] = importvelocities('velocities.txt',1,inf);
    n = size(ELEMENTS,1); %number of elements
```

```
m = size(ELEM1,1); %number of connections
[faces, n_faces] = importfaces('faces.txt', 1, inf,n);
```

Il contenuto delle variabili create in questa parte del codice è stato descritto nel §4.1, si pone ora l'attenzione sulla variabile faces non ancora descritta. Essa risulta una variabile di comodo utile solo per essere ripresa più avanti nel codice dalla funzione ffaces che si occupa di individuare le facce condivise dagli elementi con il limite del dominio.

```
VELLIQ_cell = num2cell(VELLIQ);
Xc = num2cell(X);    Yc = num2cell(Y);    Zc = num2cell(Z);
E = [ELEM1, ELEM2, VELLIQ_cell]; % alphanumeric matrix
COOR = [ELEMENTS,Xc,Yc,Zc]; % alphanumeric matrix
```

I comandi riportati sopra necessitano di una puntualizzazione su quali sono i tipi di variabili presenti in ambiente MatLab; sarà infatti necessario passare da uno all'altro per determinate operazioni. I tipi usati in questo codice sono:

- Double, variabili numeriche floating point a doppia precisione.
- String, variabili di tipo testo.
- Cell, variabili contenenti ogni tipo di informazione, sia essa di testo, numerica, o combinazione delle precedenti.

Le informazioni necessarie al codice importate dai file di input sono di tipi diversi, la maggior parte è numerica ma alcuni dati fondamentali, come ad esempio gli identificativi degli elementi, sono di tipo alfanumerico. Le regole interne di MatLab impongono ad esempio che le operazioni matematiche possano essere effettuate solo con variabili di tipo numerico, e non con variabili di tipo cell o string, anche se contenenti solo caratteri numerici. Altra regola di MatLab impone che non si possano accostare nella stessa matrice variabili di tipo diverso. Il rispetto di tali regole e la varietà dei dati utilizzati ha reso necessario comandi che creano vettori contenenti le stesse informazioni racchiuse però in variabili di tipo diverso: mediante il comando num2cell si passa ad esempio da variabili numeriche a variabili cell.

Nella parte seguente di codice si associa a ogni nodo della griglia di Voronoi un singolo valore di velocità del fluido calcolato a partire di quelli forniti da TOUGH2 alle interfacce.

Questa parte è racchiusa in un unico ciclo for, il cui indice k servirà anche oltre nel codice, che ripete le operazioni per ognuno degli n elementi che costituiscono la tassellatura di Voronoi. Il generico elemento della tassellatura oggetto dell'iterazione del ciclo for è indicato con il simbolo "C." Poiché il calcolo del campo di velocità è impegnativo dal punto di vista computazionale viene creata una barra di monitoraggio dello stato di avanzamento del ciclo for.

```
wtsect2 = waitbar(0,'Section 2 - Definition of the velocity field',
'Name', 'P3D');
    for k = 1:n
        waitbar(k / n, wtsect2);
```

Ad ogni nodo della tassellatura corrisponderà un singolo elemento di volume, si associa quindi alle coordinate del nodo l'esatto identificativo alfanumerico dell'elemento di volume che lo contiene.

```
Px = X(k,1);    Py = Y(k,1);    Pz = Z(k,1);
Pxc = num2cell(Px);
Pyc = num2cell(Py);
Pzc = num2cell(Pz);
for k1 = 1:n
    tfx = isequal(COOR(k1,2),Pxc);
    if tfx == 1
       tfy = isequal(COOR(k1,3),Pyc);
       if tfy == 1
          tfz = isequal(COOR(k1,4),Pzc);
          if tfz == 1
             C=COOR(k1,1) % C = name of the element containing
             % the node
          end
       end
    end
end
```

Seguono due cicli for che portano alla creazione della matrice denominata "I" formata da elementi di tipo cell: in questa matrice sono raccolte tutte le informazioni riguardanti le connessioni che interessano l'elemento C. La matrice I è composta da 18 colonne e un numero di righe pari a quello di blocchi confinanti con l'elemento C oggetto dell'iterazione.

Nel primo ciclo for l'elemento C viene confrontato con tutti i componenti del vettore ELEM1, che ricordiamo raccoglie il primo elemento di tutte le connessioni individuate da TOUGH2.

```
for k2 = 1:m
    tf = isequal(E(k2,1), C);
    if tf == 1
```

Si provvede a salvare una riga della matrice I per ogni corrispondenza trovata, ogni riga corrisponde quindi a una connessione che interessa l'elemento C. In colonna 1 di I viene salvato il modulo della velocità del fluido che attraversa l'interfaccia, VELLIQ, in colonna 2 l'identificativo dell'elemento C mentre in colonna 3 l'identificativo di ELEM2, secondo elemento della connessione.

```
I(i,1) = E(k2,3);
I(i,2) = E(k2,1);
I(i,3) = E(k2,2);
```

Nelle colonne 4, 5 e 6 di I vengono salvate le coordinate dell'elemento C.

```
I(i,4) = Xc(k);
I(i,5) = Yc(k);
I(i,6) = Zc(k);
```

Segue un ulteriore ciclo for per salvare nelle colonne 7, 8 e 9 di I le coordinate del secondo elemento interessato dalla connessione (ELEM2).

```
for k3 = 1:n
    tf1 = strncmpi(I(i,3),COOR(k3,1),3);
    if tf1 == 1
       I(i,7) = COOR(k3,2);
       I(i,8) = COOR(k3,3);
       I(i,9) = COOR(k3,4);
    end
end
```

In colonna 10 viene memorizzato il numero totale di facce di C (uguali per ogni riga di I), seguono una serie di colonne vuote.

```
I(i,10) = n_faces(k);
```

In colonna 18 infine è salvato un flag che segnala se l'elemento C è posto in posizione di ELEM1 o ELEM2 nella connessione.

```
I(i,18) = {1};
```

Segue il secondo ciclo for in cui l'elemento C viene confrontato con il vettore ELEM2, individuando così tutte le connessioni in cui C è il secondo elemento. Si provvede a salvare le informazioni nella matrice I (identificativi degli elementi, coordinate dei nodi, VELLIQ, numero di facce e flag sulla posizione di C nella connessione). L'architettura di questo secondo ciclo for è identica a quella del primo appena descritto.
Si ricordi che la matrice I appena costruita è costituita da variabili di tipo cell, per le regole interne di MatLab non è quindi possibili usare gli elementi di questa matrice per eseguire operazioni matematiche. Viene quindi creata la matrice "Id": in essa sono riportate le informazioni contenute nella matrice I trasformate però in variabili di tipo numerico.

```
Id(:,1) = cell2mat(I(:,1));
Id(:,4:10) = cell2mat(I(:,4:10));
Id(:,18) = cell2mat(I(:,18));
```

Questa operazione di "traduzione" fra elementi cell a elementi di tipo double non riguarda le colonne 2 e 3 in cui sono salvati gli identificativi alfanumerici degli elementi coinvolti nella connessione.
Si utilizzano ora le informazioni in forma numerica contenute nella matrice Id per le seguenti operazioni.
Il primo comando ha come obiettivo quello di trovare le componenti del vettore normale alla superficie che costituisce l'interfaccia fra due elementi. Ricordiamo che una delle conseguenze che derivano dall'aver discretizzato il dominio mediante tassellatura di Voronoi è che l'interfaccia (o il suo prolungamento) fra due elementi divide a metà il segmento congiungente i nodi ed è a esso perpendicolare. È così possibile calcolare le componenti del vettore normale all'interfaccia come differenza

delle coordinate dei nodi a cui gli elementi sono associati. Tali componenti vengono salvate nelle colonne 11, 12 e 13 di Id.

```
Id(:,11) = Id(:,7)-Id(:,4);
Id(:,12) = Id(:,8)-Id(:,5);
Id(:,13) = Id(:,9)-Id(:,6);
```

Viene calcolata la norma del vettore normale, salvata in colonna 14 di Id,

```
for k4 = 1:size(Id,1)
    Id(k4,14) = norm([Id(k4,11) Id(k4,12) Id(k4,13)]);
end
```

e successivamente il versore, dividendo le singole componenti del vettore per la norma appena trovata.

```
Id(:,15) = Id(:,11)./Id(:,14);
Id(:,16) = Id(:,12)./Id(:,14);
Id(:,17) = Id(:,13)./Id(:,14);
```

Le componenti del versore sono salvate nelle colonne 15, 16 e 17 di Id.
Il vettore rappresentato dalla colonna 1 di Id altro non è che il vettore $\boldsymbol{\gamma_i}$ descritto nella relazione (3.18), le colonne 15, 16 e 17 invece rappresentano la matrice $\boldsymbol{G_i}$ descritta nella relazione (3.17).
Si provvede a rinominare le due entità come segue.

```
gamma = Id(:,1); G=Id(:,15:17);
```

Si procede quindi alla stima della velocità di Darcy $\boldsymbol{v_i}$ come specificato nella relazione (3.20).

```
nG = ((G'*G)^-1)*G';
vi = nG*gamma;
```

Come si è potuto constatare nel Capitolo III il metodo di Painter implementato crea una distinzione fra celle “interne” e celle “di bordo” a contatto con i limiti del dominio,

utilizzando relazioni per il calcolo della velocità diverse per i due casi. Dopo aver calcolato la velocità $\widehat{\boldsymbol{v}}_{\boldsymbol{\iota}}$ si procede quindi al calcolo di $\widehat{\boldsymbol{v}}_{\iota}^{\,b}$ per le celle di bordo.
Si ricorre nuovamente alla creazione di una funzione MatLab, chiamata ffaces, che restituisce la matrice VF. Tale matrice è costituita dalle componenti dei versori normali alle facce di bordo dell'elemento C.

```
[VF] = ffaces(k, n_faces, faces, Id);
```

Gli argomenti in entrata della funzione sono l'indice k del primo ciclo for, la variabile di comodo n_faces, la matrice faces e la matrice Id.
Apriamo ora una parentesi dedicata alla creazione di VF all'interno della funzione ffaces. Per il corretto uso delle relazioni di Painter il codice deve essere in grado di distinguere gli elementi interni dagli elementi di bordo. Nei file di TOUGH2, siano essi di input o output, non è presente un flag che marchi questa differenza o che individui le facce degli elementi rivolte verso l'interno del dominio o quelle di bordo rivolte verso il suo esterno. Tali informazioni dovranno essere quindi estrapolate dal file faces.txt.
VORO2MESH nel file tough2viewer.dat fornisce il numero di facce totali dell'elemento, condivise con altri elementi o con il limite del dominio, e i versori normali che le identificano. La strategia adottata è di considerare tutti i versori normali forniti da tough2viewer.dat (facce interne più facce di bordo) e sottrarre quelli individuati utilizzando i dati di TOUGH2 (le sole facce interne). L'informazione risultante è quella dei versori delle sole facce rivolte verso l'esterno del dominio.
Il modo con cui sono state individuate le facce di confine del generico elemento C dipende dal simulatore cell-based utilizzato, nel caso in cui fossero utilizzati altri simulatori rispetto a TOUGH2 potrebbe essere necessario modificare l'architettura di ffaces.
Il punto di partenza è la matrice faces, composta da elementi di tipo cell, il primo passaggio è quello di individuare in che colonna è salvato il numero totale di facce dell'elemento. La colonna contenente l'informazione è salvata nella variabile column_faces.

```
    for k5 = 1:size(faces,2)
        tf = strncmpi(num2str(n_faces{k}),faces{k,k5},3);
        if tf == 1
            column_faces = k5;
        end
    end
```

I versori vengono quindi individuati e salvati nella matrice VF.

```
    for k6 = 1:cell2mat(n_faces(k,1))
  stringhe_versori(k6,:) =  ...
...faces(k,column_faces + cell2mat(n_faces(k,1)) + k6);
        VF(k6,:) = sscanf(stringhe_versori{k6},'(%f,%f,%f)',[1 Inf]);
    end
```

Ogni riga della matrice VF in questo momento contiene le componenti dei versori di tutte le facce dell'elemento C.

Si procede ora a cancellare da VF i versori che sono contenuti anche nelle colonne 15, 16 e 17 di Id, dove sono riportati le componenti dei versori normali alle sole facce interne dell'elemento. I versori rimanenti in VF sono quelli delle sole facce di confine.

```
    for cc = 1:size(Id,1)
        if Id(cc,18) == 1
            VF(abs(VF(:,1) - Id(cc,15)) < 0.1...
             & abs(VF(:,2) - Id(cc,16)) < 0.1...
             & abs(VF(:,3) - Id(cc,17)) < 0.1,:) = [];
        else
            VF(abs(VF(:,1) + Id(cc,15)) < 0.1...
             & abs(VF(:,2) + Id(cc,16)) < 0.1...
             & abs(VF(:,3) + Id(cc,17)) < 0.1,:) = [];
        end
    end
end
```

Viene utilizzato in questa fase il flag salvato in precedenza nella colonna 18 di I (e poi tradotto in elemento di tipo double in colonna 18 di Id) che segnalava se l'elemento C fosse ELEM1 o ELEM2. Il flag permette di determinare correttamente il segno dei versori di Id con i quali confrontare quelli salvati in VF: bisogna infatti tenere conto della convenzione dei segni del flusso, stabilito positivo da regole interne di TOUGH2 se procede da ELEM2 a ELEM1.

VF è quindi determinata, ffaces viene chiusa e la matrice creata viene importata nel Workspace di MatLab: ritorniamo quindi all'interno di P3D.

Il passo seguente è quello di calcolare il numero di facce di confine dell'elemento (pari al numero di righe di VF) e salvarlo nella variabile bordfaces. Bordfaces viene quindi salvato in colonna 19 della matrice Id.

```
bordfaces = size(VF,1);
Id(:,19) = bordfaces;
```

Segue un blocco if per il calcolo della velocità $\widehat{\boldsymbol{v}}_i^{\,b}$ per gli elementi di bordo. Viene per prima cosa costruita la matrice B contenente i versori residui di VF e in seguito il vettore $\boldsymbol{\beta}_i$ costituito da tutti elementi pari a zero. In questo modo si rappresentano le condizioni al contorno di tipo Neumann con flusso attraversante le facce al contorno pari a 0.

```
if bordfaces ~= 0
    B = VF(:,1:3);
    beta = zeros(size(B,1),1);
```

La velocità $\widehat{\boldsymbol{v}}_i^{\,b}$ viene calcolata con la relazione (3.22).

```
    nGB = ((G'*G)^-1)*B'*((B*((G'*G)^-1)*B')^-1);
    vib = vi-nGB*(B*vi-beta);
end
```

Si crea infine la matrice M in cui sono raccolte le coordinate dei nodi Px, Py e Pz (salvate nelle colonne 1, 2 e 3) e i singoli valori di velocità ad essi associati (colonne 4, 5 e 6). Se l'elemento C considerato è interno si provvede a salvare il valore $\widehat{\boldsymbol{v}}_i$, se è elemento di bordo $\widehat{\boldsymbol{v}}_i^{\,b}$.

```
M(k,1:3) = [Px Py Pz];
if bordfaces == 0
    M(k,4:6) = -vi';
else
    M(k,4:6) = -vib';
end
```

Si provvede a pulire le matrici I e Id e a concludere il ciclo for iniziale.

```
    clear Id I
end
```

La matrice M contiene il campo di velocità dell'intero dominio, viene quindi salvata nel file di testo denominato velocityfield.txt. La disposizione delle informazioni nel file è illustrata nel §4.1.5.

```
    % Velocity field is saved in a txt file
    filename = ['velocityfield.txt'];
    fid=fopen(filename,'w');
    fprintf(fid, [ 'X            ' 'Y            ' 'Z            '...
                   'VX           ' 'VY           ' 'VZ           \n']);
    fprintf(fid, '%.5e %.5e %.5e %.5e %.5e %.5e \n', ...
        [M(:,1) M(:,2) M(:,3) M(:,4) M(:,5) M(:,6)]');
    fclose(fid);
    close(wtsect2);
end

% ######################## END OF SECTION 2 ##########################
```

Si chiude in questo modo la Section 2 del codice in cui viene definito il campo di velocità del dominio, per importazione (prima modalità e primo blocco dell'istruzione if) o per calcolo (seconda modalità e secondo blocco dell'if).

4.3.3 Section 3, generazione della griglia di Delaunay

La Section 3 è costituita da un solo comando: sulla base dei nodi specificati nelle colonne 1, 2 e 3 della matrice M viene creata la tassellatura di Delaunay del dominio.

```
%%
% ############################ SECTION 3 ############################
% CREATION OF DELAUNAY TESSELLATION

T = delaunayn(M(:,1:3));
```

Il risultato ottenuto è la matrice T in cui è rappresentato per ogni riga un tetraedro costituente la tassellatura di Delaunay, sono infatti riportati gli indici di riga della matrice M in cui sono specificate le coordinate dei quattro vertici del tetraedro.

4.3.4 Section 4, calcolo delle traiettorie

La Section 4 dispone di una seconda barra di avanzamento che monitora in questo caso lo stato di avanzamento del calcolo delle traiettorie.

```
wtsect4 = waitbar(0,'Section 4 - Pathline tracing','Name','P3D');
```

La quasi interezza della Section 4 è contenuta all'interno di un ciclo for che ripete le operazioni per ognuno dei punti di partenza delle traiettorie specificati nella matrice seed.

Le traiettorie vengono memorizzate in questa fase all'interno di tre variabili diverse XP, YP e ZP. Il punto definito nella prima riga di seed individua la prima traiettoria calcolata, che chiameremo Traiettoria 1; questa sarà composta dalle posizioni occupate dalla particella fluida a distanza dell'intervallo di tempo dt. La coordinata x di tali posizioni sarà salvata in colonna 1 di XP, la coordinata y in colonna 1 di YP e infine la coordinata z in colonna 1 di ZP. Il procedimento si applica tal quale per quella che sarà Traiettoria 2, le cui posizioni svengono salvate in colonna 2 delle variabili XP, YP e ZP: si procede in questo modo per ognuna delle traiettorie cercate.

Il primo elemento di XP, YP e ZP, e quindi prima posizione della traiettoria, è estratto dalla matrice seed.

```
for n_seed = 1:size(seed)
    XP(1,n_seed) = seed(n_seed,1);
    YP(1,n_seed) = seed(n_seed,2);
    ZP(1,n_seed) = seed(n_seed,3);
```

Per il calcolo delle posizioni seguenti si fa uso di un ciclo while che si ripete finché non interviene un criterio di arresto dato dal valore della variabile CDA.

Per CDA pari a 0 viene compiuto un ulteriore ciclo, per CDA pari a 1 si passa al punto di seed successivo.

```
    n_it = 2; % Number of iterations
    CDA = 0; % Stop criterion
    while CDA == 0;
```

La prima operazione del ciclo while è quella di definire la variabile P come l'ultima posizione nota della traiettoria.

```
P = [XP(n_it-1,n_seed) YP(n_it-1,n_seed) ZP(n_it-1,n_seed)];
```

Con un unico comando di MatLab si ottengono l'elemento della tassellatura di Delaunay contenente P (ti) e le coordinate baricentriche [28] di P rispetto ai vertici di ti (bc).

```
[ti, bc] = tsearchn(M(:,1:3),T,P);
```

Viene quindi verificata la prima condizione che, se presente, porta all'interruzione del ciclo while. Se l'elemento di Delunay ti assume un valore non numerico, NaN, CDA viene posto pari a 1 e il codice passa al calcolo della traiettoria dal punto di seed successivo.

```
if isnan(ti)
    CDA = 1
```

Se tale condizione non si verifica il ciclo while continua e vengono quindi definite le velocità v1, v2, v3 e v4 come le velocità assegnate in Section 3 ai quattro vertici dell'elemento di Delaunay ti.

```
else
v1 = M(T(ti,1), 4:6);
v2 = M(T(ti,2), 4:6);
v3 = M(T(ti,3), 4:6);
v4 = M(T(ti,4), 4:6);
```

Le tre componenti vx, vy e vz della velocità del fluido nel punto P sono quindi calcolate come media pesata fra i valori v1, v2, v3 e v4. I pesi utilizzati sono le coordinate baricentriche bc.

```
vx = v1(1)*bc(1) + v2(1)*bc(2) + v3(1)*bc(3) + v4(1)*bc(4);
vy = v1(2)*bc(1) + v2(2)*bc(2) + v3(2)*bc(3) + v4(2)*bc(4);
vz = v1(3)*bc(1) + v2(3)*bc(2) + v3(3)*bc(3) + v4(3)*bc(4);
```

I valori vx, vy e vz sono salvati in tre distinte matrici VX, VY e VZ.

```
VX(n_it,n_seed) = vx;
VY(n_it,n_seed) = vy;
VZ(n_it,n_seed) = vz;
```

Viene ora calcolato l'intervallo temporale con cui andremo a moltiplicare le componenti della velocità. Per ricavarlo si divide lo spostamento minimo stabilito in Section 1 per la norma del vettore velocità ottenendo in questo modo un intervallo temporale. Gli intervalli temporali vengono salvati in forma di matrice.

```
dt(n_it,n_seed) = ds_min / (vx^2 + vy^2 + vz^2)^0.5;
```

Le componenti dello spostamento sono calcolate come prodotto fra le componenti di velocità e l'intervallo di tempo appena trovato.

```
sx = vx*dt(n_it,n_seed);
sy = vy*dt(n_it,n_seed);
sz = vz*dt(n_it,n_seed);
```

La nuova posizione della particella fluida è calcolata come somma fra la posizione precedente e lo spostamento.

```
XP(n_it,n_seed) = XP(n_it-1,n_seed) + sx;
YP(n_it,n_seed) = YP(n_it-1,n_seed) + sy;
ZP(n_it,n_seed) = ZP(n_it-1,n_seed) + sz;
```

Vengono poi verificate le altre condizioni di arresto: ossia se la posizione finale è dentro l'intorno della posizione finale presunta o se il numero di iterazioni ha superato il numero massimo imposto in Section 1. Se una o più di queste condizioni sono verificate il ciclo while si arresta, se invece non si verificano si procede al calcolo di una ulteriore posizione.

```
if abs(XP(n_it,n_seed)-XF) < tolx && ...
   abs(YP(n_it,n_seed)-YF) < toly && ...
   abs(ZP(n_it,n_seed)-ZF) < tolz
   CDA = 1;
elseif n_it > n_it_max
    CDA = 1;
else
    CDA = 0;
```

```
        end
```

Si calcola il tempo di volo associato a ogni singola posizione della traiettoria e si fa avanzare il contatore delle iterazioni eseguite di una unità.

```
        TOF(n_it,n_seed) = sum(dt(:,n_seed));
        n_it = n_it + 1;
```

Vengono chiusi l'istruzione if e il ciclo while, si calcola lo stato di avanzamento rappresentato con la barra aperta all'inizio della Section e si chiude il ciclo for.

```
        end
    end
    waitbar(n_seed / size(seed,1), wtsect4);
end
```

Una volta conclusa la fase di calcolo delle traiettorie viene chiusa la barra di avanzamento.

```
close(wtsect4);
```

4.3.5 Section 5, visualizzazione e salvataggio delle traiettorie

Le traiettorie sono a questo punto salvate nelle matrici XP, YP e ZP. In Section 5 il codice crea una figura in cui tutte le traiettorie sono visualizzate in un unico grafico; la colorazione dei vari punti avviene in base alla coordinata z.

```
%%
% ############################ SECTION 5 ##############################
% PATHLINES DISPLAY AND SAVE

for k = 1:size(seed,1)
    scatter3(XP(:,k),YP(:,k),ZP(:,k),2,ZP(:,k))
    hold on
```

La figura non viene automaticamente salvata, tale operazione può essere fatta direttamente dall'utente nel formato preferito.

Il codice prevede inoltre di salvare un file di testo per ognuna delle traiettorie: il contenuto di questo file è descritto nel §4.5.

```
    filename = ['Pathline' num2str(k) '.txt'];
    fid=fopen(filename,'w');
    fprintf(fid, [ 'X[m]    ' 'Y[m]    ' 'Z[m]     ' 'TOF[sec]   '...
                   'VX[m/s] ' 'VY[m/s  ' 'VZ[m/s]  ' '\n']);
    for k4 = 1:size(XP(:,k),1)
        if ~isnan(XP(k4,k))
            k_stop(k) = k4;
        end
    end


fprintf(fid, '%f %f %f %d %e %e %e \n', ...
      [XP(1:k_stop,k) YP(1:k_stop,k) ZP(1:k_stop,k) TOF(1:k_stop,k)...
       VX(1:k_stop,k) VY(1:k_stop,k) VZ(1:k_stop,k)]');
    fclose(fid);
end
```

Sul grafico, in prossimità dei punti d'inizio delle traiettorie, viene visualizzato un numero progressivo. La prima traiettoria verrà calcolata a partire dal punto descritto nella prima riga della matrice seed, a fianco della linea che descrive tale traiettoria nel grafico sarà quindi posto il numero "1"; la numerazione prosegue così per tutte le traiettorie calcolate.

```
for k = 1:size(seed,1)
    text(XP(1,k), YP(1,k), ZP(1,k), num2str(k));
end
```

Vengono aggiunti al grafico il titolo principale e le etichette agli assi.

```
title('Pathline display')
xlabel('X  [m]'); ylabel('Y  [m]'); zlabel('Z  [m]');
```

È impostato che il volume racchiuso dagli assi si adatti perfettamente ai dati rappresentati dal grafico.

```
axis image
```

Vengono infine aggiunte le griglie primarie e secondarie e la legenda che descrive la colorazione delle traiettorie.

```
grid on
grid minor
cbt = colorbar;
xlabel(cbt,'Z-coordinate  [m]')
hold off

% ######################## END OF SECTION 5 ##########################
% ######################## END OF THE CODE ###########################
```

4.4 P2D

P2D nasce dall'esigenza di poter agire su griglie monostrato non strutturate usate per discretizzare domini 2D o 2.5D, ossia quelli in cui per tutte le coppie di coordinate X e Y corrisponde un singolo valore della coordinata Z [4]. Tali domini sono stati scorporati dall'ambito di applicazione di P3D al fine di non rendere quest'ultimo inutilmente più complesso.

L'architettura di P2D è sostanzialmente uguale a quella del suo corrispettivo tridimensionale, viene però cambiata la gestione della coordinata Z in tutte le funzioni associate al codice.

4.5 I file di output

Il codice produce in output, come anticipato, un'immagine e una serie di file di testo. L'immagine è creata ma non viene salvata in automatico dal codice, provvederà infatti l'utente a farlo nel formato preferito. Il formato nativo delle immagini in MatLab è il .fig, salvando l'immagine con questa estensione non abbiamo alcuna perdita di informazioni; è anche possibile salvare l'immagine in formati più comuni come il .jpg o il .png.

Al contrario dell'immagine i file di testo sono salvati dal codice stesso.

Nel caso in cui il campo di velocità dovesse essere calcolato dal codice a partire dai dati della simulazione numerica il primo file salvato è velocityfield.txt; di tale file e del suo doppio "ruolo" di file di input e output si è parlato nel §4.1.5.

Qualsiasi sia la modalità per fornire i dati di input scelta dall'utilizzatore, il codice crea dei file di testo in cui sono memorizzate le traiettorie, denominati con ordine progressivo "Pathline1.txt", "Pathline2.txt", etc. L'organizzazione delle informazioni nelle file è quella mostrata in Figura 4.7: le prime tre colonne sono dedicate alle coordinate dei punti che costituiscono la traiettoria, la quarta colonna è dedicata al tempo di volo impiegato dalla particella a raggiungere la posizione corrispondente mentre nelle ultime colonne sono riportate le componenti del vettore velocità della particella.

X[m]	Y[m]	Z[m]	TOF[sec]	VX[m/s]	VY[m/s]	VZ[m/s]
50.000000	50.000000	-50.000000	0	0.000000e+00	0.000000e+00	0.000000e+00
50.577275	50.577275	-50.577500	7.754386e+05	7.444500e-07	7.444500e-07	-7.447400e-07
51.154128	51.155394	-51.154579	1.560399e+06	7.348805e-07	7.364942e-07	-7.351692e-07
51.730544	51.734381	-51.731224	2.355119e+06	7.253070e-07	7.285413e-07	-7.255945e-07
52.306510	52.314260	-52.307421	3.159846e+06	7.157296e-07	7.205913e-07	-7.160158e-07
52.882013	52.895057	-52.883156	3.974835e+06	7.061480e-07	7.126443e-07	-7.064330e-07

Figure 4.7– Come si presenta il file Pathline.txt.

Capitolo V

Verifica e Validazione

In fine i codici (il P2D e il P3D) sono stati sottoposti ad un approfondito controllo utilizzando domini bidimensionali e tridimensionali. Questa analisi ha lo scopo di verificare e validare i due codici prodotti. Sono stati valutati il corretto calcolo delle traiettorie, in particolare per i casi dove è possibile il confronto con una soluzione analitica, e il tempo impiegato per il loro calcolo.

5.1 Test bidimensionali

Il caso bidimensionale considerato per testare il codice è un classico dell'industria petrolifera, si tratta infatti del cosiddetto schema a cinque punti (*five spot* nella letteratura anglosassone) mostrato in Figura 5.1.

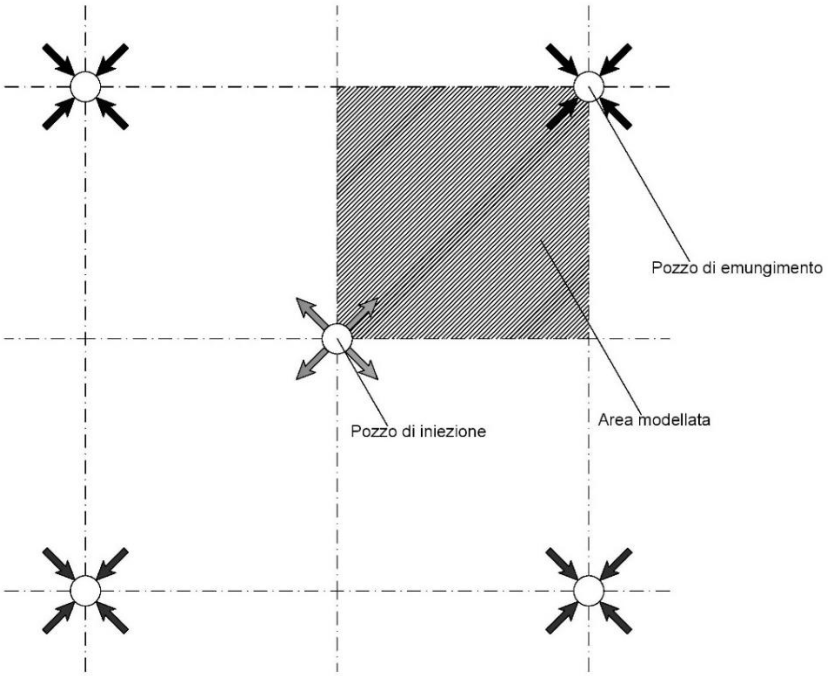

Figura 5.1 – Schema five spot.

Tale schema, ripetuto infinite volte nel piano, è caratterizzato da quattro pozzi di emungimento posti ai vertici di un quadrato mentre il pozzo di iniezione è posto nel suo centro. Data la simmetria dello schema, per il suo studio ci si può limitare ad analizzare solamente un quarto del quadrato.

Per descrivere l'andamento di un esperimento di emungimento-iniezione del five spot è disponibile anche la soluzione analitica calcolata da Morel-Seytoux [20] [22], quest'ultima sarà utilizzata per confrontare i risultati del codice.

5.1.1 Griglia strutturata

Si è inizialmente indagato un dominio 500 × 500 m^2 discretizzato mediante una griglia strutturata 11 × 11. È stato scelto di avere i nodi della griglia posti sul perimetro esterno del dominio affinchè la triangolazione di Delaunay fosse coincidente con il dominio di simulazione.

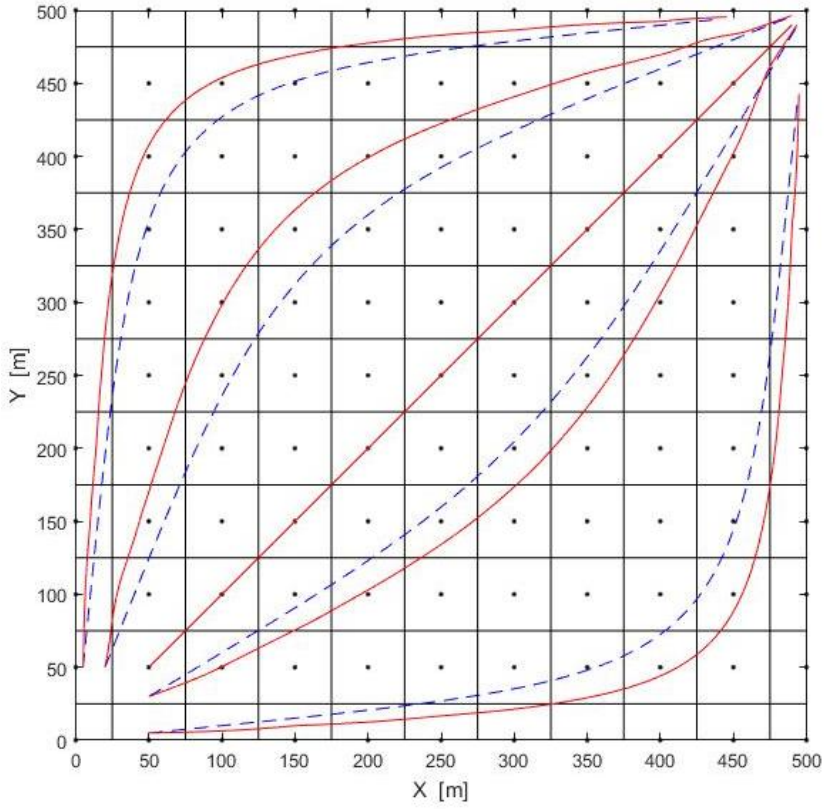

Figura 5.2 – Soluzione di P2D (linea continua rossa) messa a confronto con la soluzione analitica (linea tratteggiata blu) su griglia strutturata da 121 elementi.

In Figura 5.2, sono mostrate le traiettorie ottenute con P2D (linea continua rossa) e quelle ottenute con la soluzione analitica (linea tratteggiata blu).

Pur essendo la griglia discretizzara con pochi blocchi l'andamento generale del flusso è correttamente individuato: la traiettoria centrale si sovrappone perfettamente a quella della soluzione analitica e le due traiettorie periferiche rimangono costantemente dentro l'area modellata. Cionostante le differenze fra la soluzione del codice e la soluzione analitica sono consistenti. A spiegazione di questo aspetto si fa notare che la stessa soluzione cell-based è influenzata dal grado di raffittimento della griglia utilizzata per la discretizzazione del dominio, ne consegue che la discrepanza fra le traiettorie non è imputabile all'approccio discreto necessariamente implementato nel codice o al metodo di Painter, ma dipende principalmente dai dati di input utilizzati. Griglie strutturate più raffittite portano a risultati più precisi. A riprova di ciò nel successivo test 2D il numero di elementi della griglia viene aumentato a 676, i risultati sono riportati in Figura 5.3.

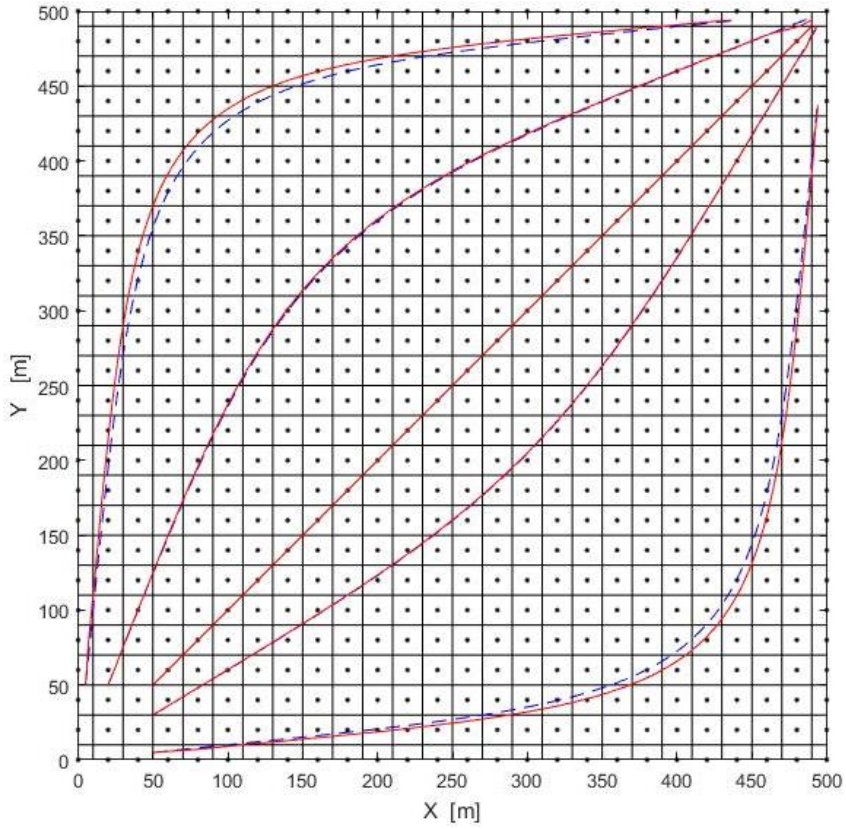

Figura 5.3 – Soluzione di P2D (continua rossa) messa a confronto con la soluzione analitica (tratteggiata blu) su griglia strutturata da 676 elementi.

Le traiettorie centrali si sovrappongono alla soluzione analitica mentre le discrepanze nelle traiettorie esterne sono minime.

5.1.2 Griglia non strutturata

Lo step successivo è stato contemplare una griglia non struttrata avente anche numero di elementi circa doppio rispetto a quella del primo esempio (precisamente 243). Si noti come la forma degli elementi è spiccatamente irregolare. I risultati ottenuti sono mostrati in Figura 5.4, le differenze fra soluzione prodotta dal codice P2D e quella ottenuta con la soluzione analitica sono drasticamente calate.

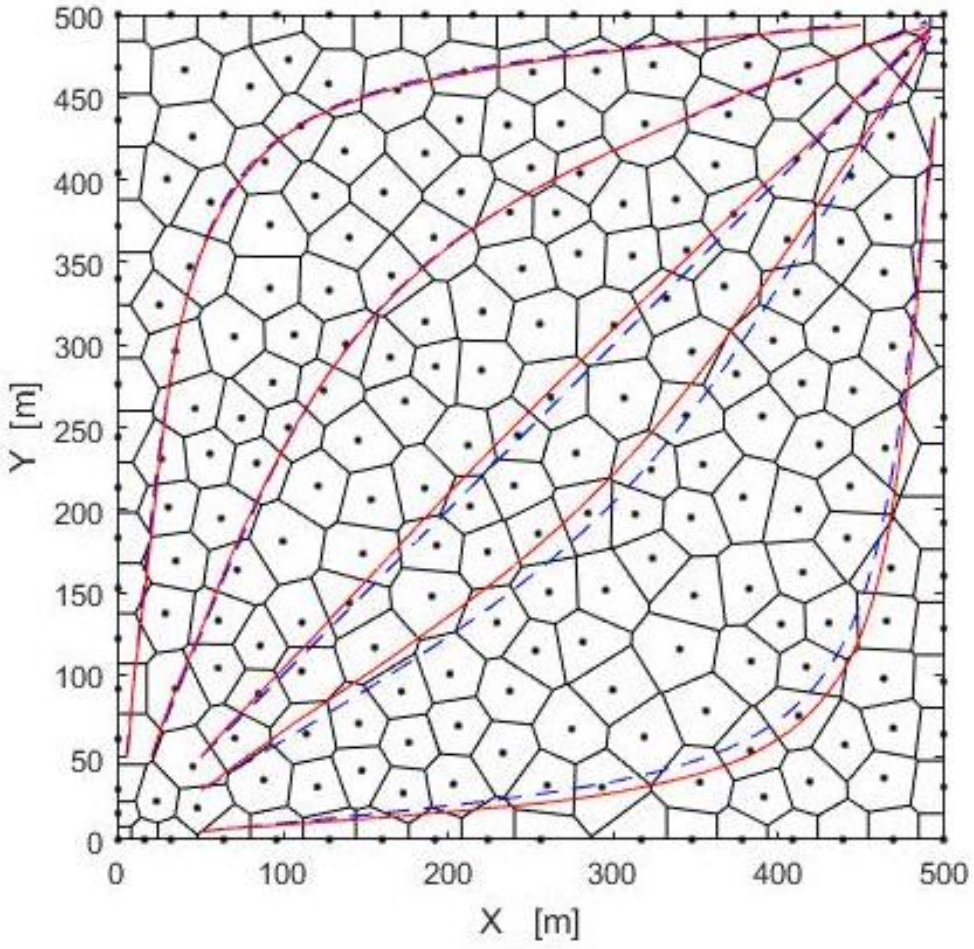

Figura 5.4 – Soluzione di P2D (in rosso) messa a confronto con la soluzione analitica (in tratteggiata blu) su griglia non strutturata da 243 elementi

5.1.3 Tempi di calcolo

La precisione nel calcolo delle traiettorie non era però l'unico obiettivo di questo test. Si puntava infatti a valutare quanto fosse perfomante P2D. In particolare sono state eseguite due prove in cui il tempo di calcolo è stato misurato con la funzione *Run and time* di MatLab, nella prima il campo di velocità deve essere calcolato dal codice, nella seconda invece è già disponibile nel file velocityfiled.txt.

La voce "Self Time" nelle tabelle 5.1 e 5.2 individua il tempo impiegato dalla CPU nell'eseguire la funzione a cui si riferisce escludendo le funzioni che sono richiamate al suo interno.

Function Name	Calls	**Total Time**	Self Time*	Total Time Plot (dark band = self time)
P2D	1	8.680 s	5.561 s	
tsearchn	6846	1.719 s	0.198 s	
polyfun\private\tsrchnmx	6846	1.521 s	1.521 s	
ffaces	243	1.011 s	0.184 s	

Tabella 5.1 – Tempo di calcolo di P2D con calcolo del campo di velocità, dominio di 243 elementi e 10 traiettorie.

Function Name	Calls	**Total Time**	Self Time*	Total Time Plot (dark band = self time)
P2D	1	2.914 s	1.035 s	
tsearchn	6846	1.748 s	0.202 s	
polyfun\private\tsrchnmx	6846	1.547 s	1.547 s	
newplot	1	0.055 s	0.041 s	

Tabella 5.2 – Tempo di calcolo di P2D con campo di velocità già calcolato, dominio di 243 elementi e 10 traiettorie.

Con un PC equipaggiato con CPU Intel da 2.6 GHz e 12 GB di Ram l'esecuzione di P2D sul dominio da 243 elementi, nel caso di 10 traiettorie, ha impiegato i tempi riportati nelle Tabelle 5.1 e 5.2. La parte computazionalmente più gravosa del codice riguarda il calcolo del campo di velocità.

I tempi di calcolo sono influenzati da diversi fattori oltre che dalle caratteristiche del PC e dal numero di elementi e traiettorie: in particolar modo si menzionano lo spostamento minimo ds_min e la lunghezza della. Di tali fattori non si può tenere conto puntualmente, il valore misurato può però essere utilizzato come un riferimento dell'efficienza computazionale del codice.

5.2 Test tridimensionali

Per i test tridimensionali non sono disponibili soluzioni di tipo analitico, i risultati del codice P3D devono essere quindi valutati sulla base della plausibilità fisica. Sono state utilizzate due griglie, una strutturata e una non struttrata con numero uguale di elementi.

Il dominio tridimensionale su cui è stato testato P3D è un cubo di dimensioni 500 × 500 × 500 m^3 con coordinata z che si sviluppa lungo il semiasse negativo. Come mostrato in Figura 5.5, sono inseriti un elemento di tipo *source* nel blocco di coordinate (1.000E-03, 1.000E-03, -1.000E-03) e un elemento di tipo *sink* con coordinate (500, 500, -500).

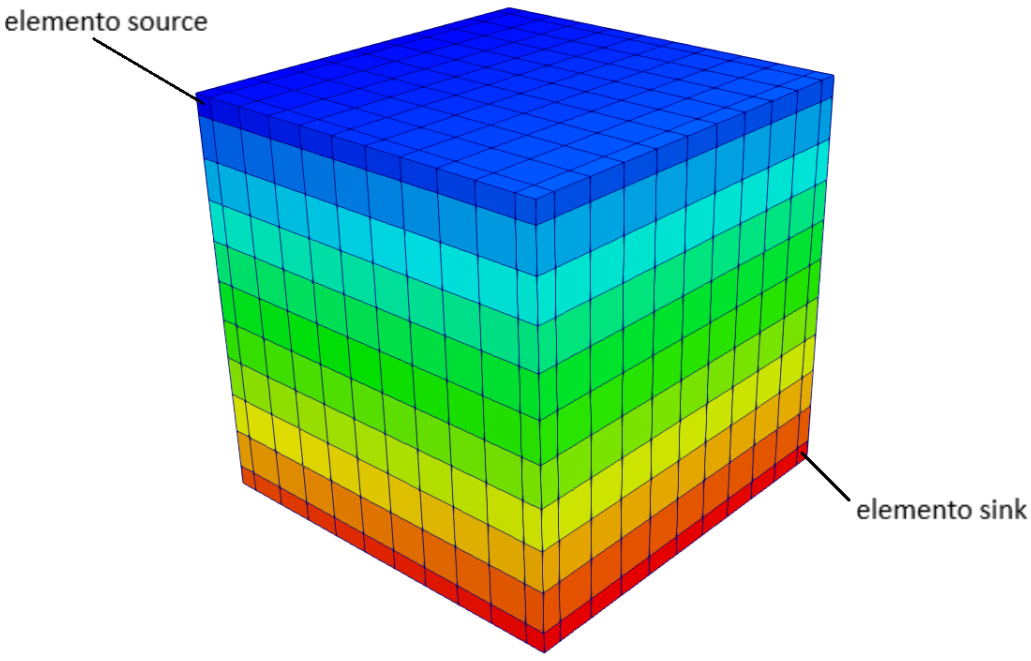

Figura 5.5 – Posizione degli elementi source e sink, griglia strutturata.

5.2.1 Griglia strutturata

La prima discretizzazione utilizzata è strutturata con 11 × 11 × 11 (1331) elementi, sono prese in considerazione tre traiettorie a partire dall'intorno della cella source che si sviluppano come mostrato nella sezione 2D in Figura 5.5 fino a raggiungere la cella sink. Come anticipato nel §4.3.5 la colorazione delle traiettorie avviene in base al valore assunto dalla coordinata Z.

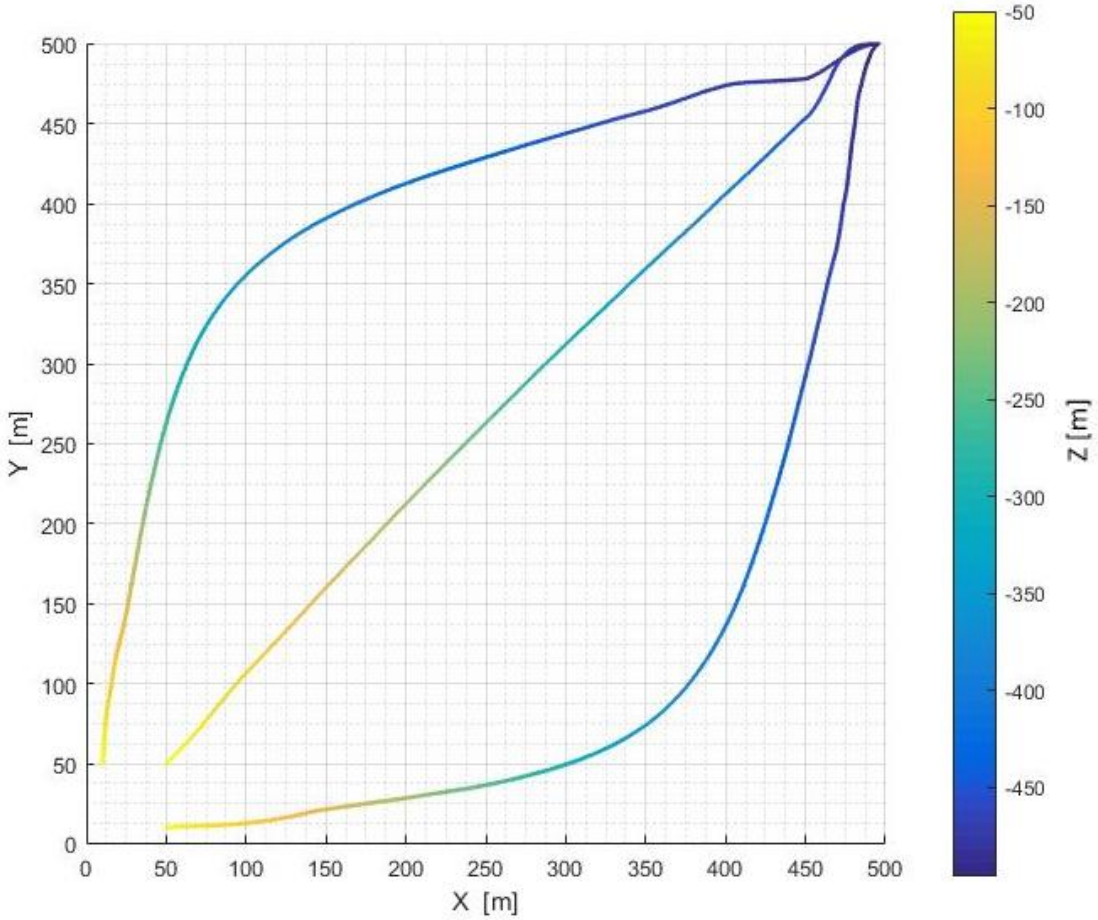

Figura 5.6 – Soluzione di P3D su dominio cubico, griglia strutturata da 1331 elementi, sezione 2D corrispondente al piano z=0.

5.2.2 Griglia non strutturata

Sullo stesso dominio è stata eseguita una ulteriore simulazione con griglia non strutturata, con egual numero di blocchi (1331) e con un numero variabile di facce da 6 a 18. La suddivisione del dominio è riportata in Figura 5.7.

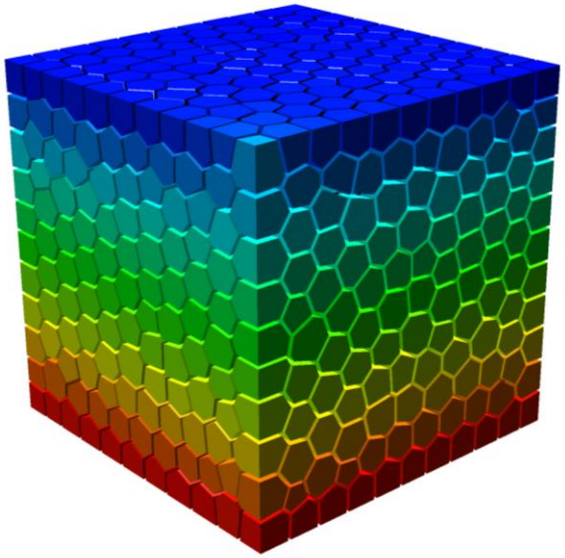

Figura 5.7 – Discretizzazione su dominio cubico, griglia non strutturata da 1331 elementi.

Le soluzioni di P3D ottenute sulle due griglie sono poste a confronto in Figura 5.8: le differenze che si possono notare dipendono dalla diversa costruzione del campo di velocità discretizzato.

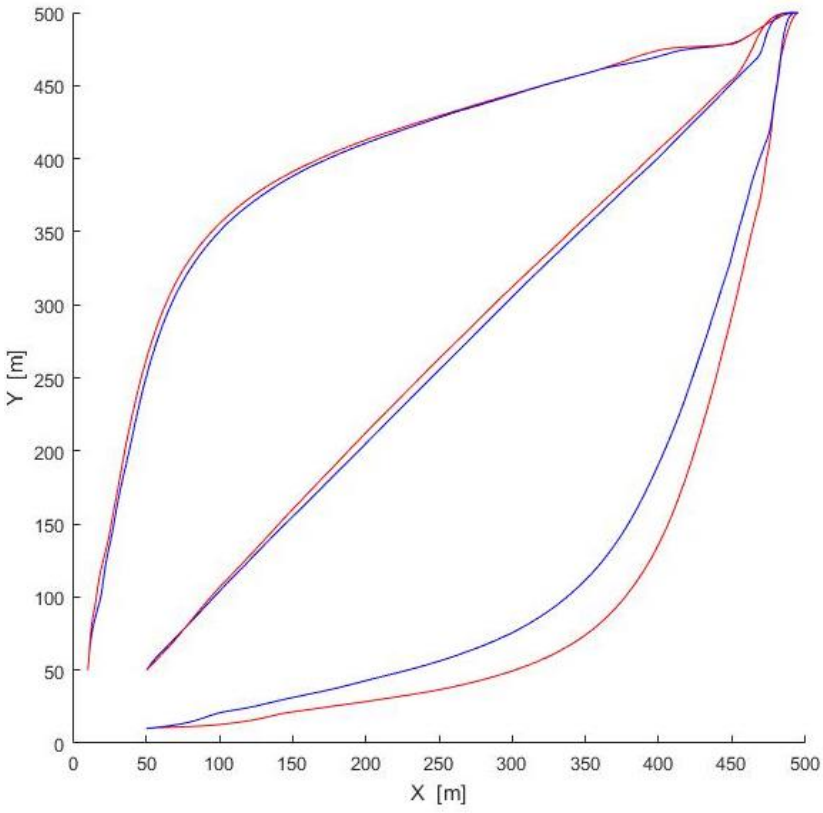

Figura 5.8 – Confronto soluzione su griglia strutturata (in rosso) e non strutturata (in blu).

5.2.3 Tempi di calcolo

Per quanto riguarda i tempi di calcolo anche per P3D sono state eseguite due prove con campo di velocità da calcolare o meno. I risultati ottenuti, visibili nelle Tabelle 5.3 e 5.4, confermano che il calcolo del campo di velocità è la parte del codice che impegna per più tempo la CPU.

Function Name	Calls	**Total Time**	Self Time*	Total Time Plot (dark band = self time)
P3D	1	264.025 s	246.098 s	
tsearchn	2565	8.400 s	0.134 s	
polyfun\private\tsrchnmx	2565	8.266 s	8.266 s	
ffaces	1331	5.356 s	1.043 s	
num2str	139764	4.210 s	1.802 s	
int2str	139764	2.408 s	2.408 s	

Tabella 5.3 – Tempo di calcolo di P3D con calcolo del campo di velocità, dominio di 1331 elementi e 3 traiettorie.

Function Name	Calls	**Total Time**	Self Time*	Total Time Plot (dark band = self time)
P3D	1	10.361 s	0.929 s	
tsearchn	2565	8.541 s	0.137 s	
polyfun\private\tsrchnmx	2565	8.404 s	8.404 s	
close	1	0.339 s	0.001 s	
close>request_close	1	0.277 s	0.005 s	
colorbar	1	0.271 s	0.011 s	

Tabella 5.4 – Tempo di calcolo di P3D con campo di velocità già calcolato, dominio di 1331 elementi e 3 traiettorie.

5.2.4 Dominio con barriera quasi totalmente impermeabile

Nell'ultimo test a cui si sottopone il codice il dominio è nuovamente discretizzato il 1331 elementi con una griglia strutturata, viene però aggiunta una barriera quasi totalmente

impermeabile con una “finestra” centrale permeabile che permette un passaggio obbligato del fluido. La barriera è mostrata in Figura 5.9.

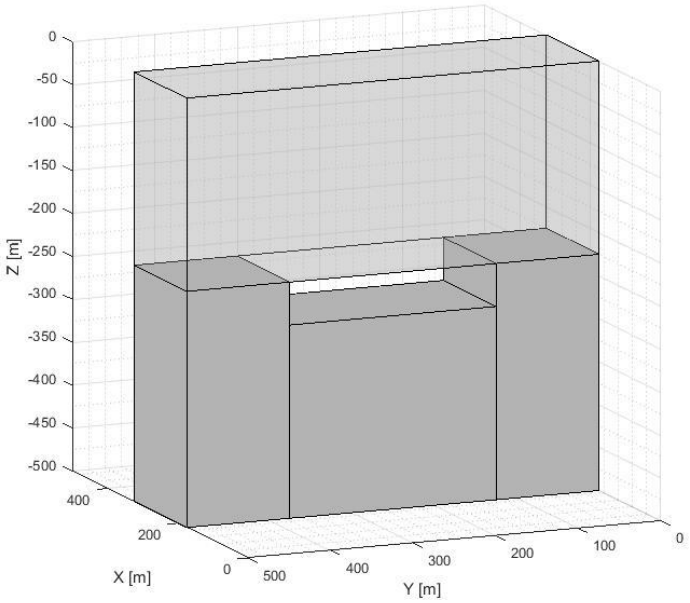

Figura 5.9 – Barriera attraverso cui il fluido è forzato a passare. Il blocco superiore è reso in trasparenza.

I risultati ottenuti sono presentati nelle Figure 5.11, 5.12 e 5.13, le traiettorie attraversano la finestra permeabile e sono fisicamente plausibili.

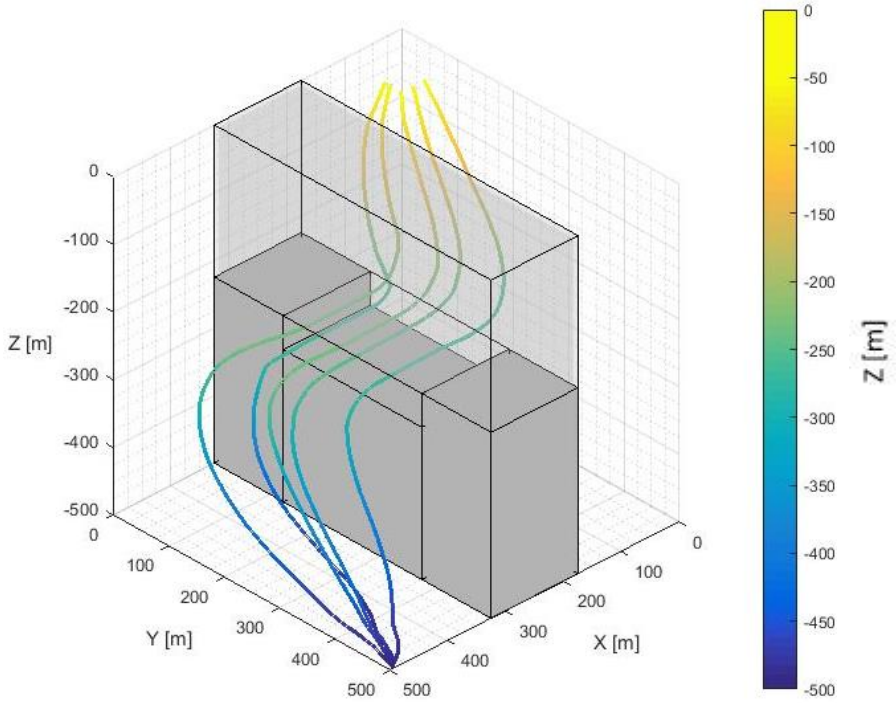

Figura 5.10 – Traiettorie ottenute con P3D, visuale prospettica.

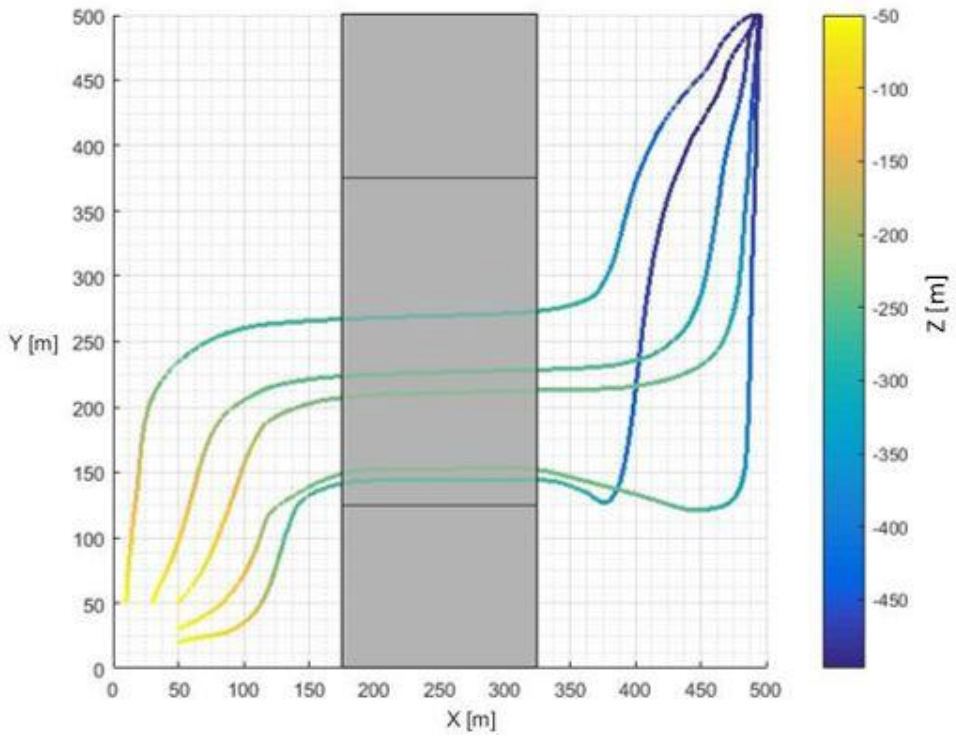

Figura 5.11 – Traiettorie ottenute, vista nella sezione 2D ottenuta nel piano z=0.

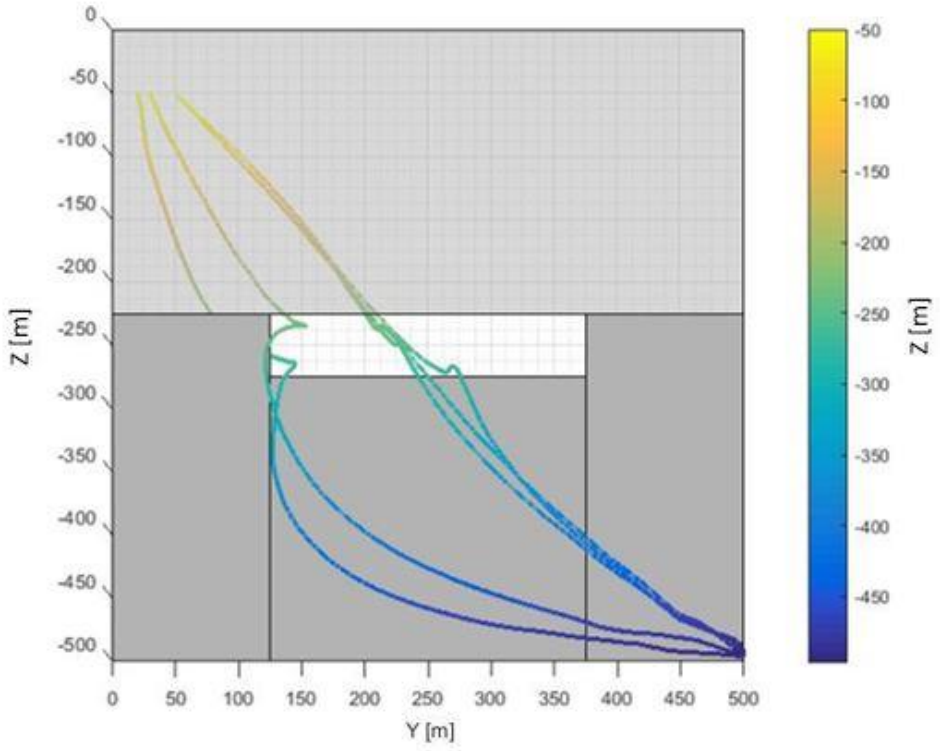

Figura 5.12 – Traiettorie ottenute, vista nella sezione 2D ottenuta nel piano x=0 dal lato dell'elemento sink.

Capitolo VI

Conclusioni e futuri sviluppi

L'obiettivo di questa tesi è la realizzazione di uno strumento informatico per la generazione di traiettorie di un fluido nel suo moto all'interno di un mezzo poroso. Tale strumento deve poter operare su griglie non strutturate, cioè con griglie caratterizzate da elementi di forma irregolare.

Tali esigenze vengono soddisfatte dal metodo di calcolo scelto, proposto da Painter [23] e illustrato nel Capitolo III.

Dal punto di vista della fruibilità si è puntato invece a rendere lo strumento facilmente accessibile e veloce sia nella fase di preprocessing dei dati che in quella di calcolo.

L'ambiente di sviluppo scelto è stato MatLab, il codice elaborato è riportato e spiegato nel Capitolo IV. Le operazioni di preprocessing sono limitate mentre il settaggio del codice da parte dell'utente è intuitivo.

Il codice calcola e mostra le traiettorie a partire dai dati forniti da una simulazione cell-based che genera il campo delle velocità.

I test eseguiti su diversi domini bi e tridimensionali discretizzati mediante griglie strutturate e non strutturate hanno dimostrato l'affidabilità e la piena funzionalità del codice anche con griglie a diverso grado di raffittimento o caratterizzate da una forma irregolare dei blocchi.

Si sono infine dimostrati i vantaggi in termini di precisione ottenuti dall'uso di griglie a maglia fine e, come atteso, l'influenza che la forma degli elementi ha sul risultato finale.

La direzione principale verso cui si intende far evolvere il codice è sicuramente quella di una maggiore automazione del suo utilizzo. Come si è visto nel Capitolo IV l'importazione dei dati in ambiente MatLab necessari alla corretta esecuzione del codice, necessita di file intermedi (elements.txt, velocities.txt) ricavati dai file di output

del simulatore cell-based. Un possibile sviluppo è quello di costruire dei filtri specifici in grado di importare i dati necessari al codice direttamente dai file di output del simulatore eliminando il passaggio attraverso i file intermedi che, come abbiamo visto, necessitano di operazioni di preprocessing da parte dell'utente.

Un altro sviluppo potrebbe riguardare la gestione della condizione di flusso nullo ai confini del dominio, potenzialmente superabile nei metodi streamline-based ma mantenuta invece nel presente codice. Mediante nuove function potrebbero essere importati dai file forniti dal simulatore cell-based i flussi diversi da zero ai confini. In questo modo l'analisi compiuta dal codice non dovrebbe necessariamente riguardare l'intero dominio della simulazione cell-based, ma si potrebbe limitare a specifici settori di maggiore interesse.

Si intende infine sviluppare il codice del codice affinché risulti utilizzabile nella sua interezza all'interno di software freeware compatibili a MatLab come Octave [35].

Appendice A

P3D

```
%
########################################################################
%NAME     :  P3D.m
%PURPOSE  :  Pathline tracing on fully unstructured Voronoi 3D grid
%AUTHOR   :  Matteo Scapolo
%
%INPUT 1  :  seed.txt,velocityfield.txt
%INPUT 2  :  seed.txt,elements.txt,velocities.txt,faces.txt
%OUTPUT   :  Pathline display, Pathline1.txt, Pathline2.txt, etc.
%
########################################################################
clear all
clc
close all
%%
% ############################## SECTION 1 ##############################
% DEFINITION OF THE PARAMETERS BY THE USER

% Min. displacement
ds_min = 1; % meters

% Max. iterations
n_it_max = 10^5;

% Final position
XF = 500;     YF = 500;     ZF = -500; %meters

% Final position tolerances
tolx = 5;     toly = 5;     tolz = 5; %meters

% Velocity field is already defined? 0 for NO, 1 for YES
VField = 0;

% Pathline seeds import
[seed] = importseed('seed.txt', 1, inf)

% ######################### END OF SECTION 1 ############################
%%
% ############################## SECTION 2 ##############################
% DEFINITION OF THE VELOCITY FIELD

if VField == 1; % Velocity field is already defined
   M = importvelocityfield('velocityfield.txt', 2, inf);
else % VField == 0;  % Velocity field is NOT defined

    % IMPORT DATA
    [ELEMENTS,X,Y,Z] = importelements('elements.txt',1,inf);
    % ELEMENTS = Element names
```

```
    % X,Y,Z    = Node coordinates
    [ELEM1,ELEM2,VELLIQ] = importvelocities('velocities.txt',1,inf);
    % ELEM1  = Fist element of the connection
    % ELEM2  = Second element of the connection
    % VELLIQ = Velocity of the fluid at the interface
    n = size(ELEMENTS,1); % number of elements
    m = size(ELEM1,1); % number of connections
    [faces, n_faces] = importfaces('faces.txt', 1, inf,n);
    % faces   = Passing variable
    % n_faces = Number of total faces of the element

    % Conversion of numerical vectors in alphanumeric vectors
    VELLIQ_cell = num2cell(VELLIQ);
    Xc = num2cell(X);    Yc = num2cell(Y);    Zc = num2cell(Z);
    E = [ELEM1, ELEM2, VELLIQ_cell]; % alphanumeric matrix
    COOR = [ELEMENTS,Xc,Yc,Zc]; % alphanumeric matrix

    % VELOCITIES ASSIGNMENT TO VORONOI NODES
wtsect2 = waitbar(0,'Section 2 - Definition of the velocity field',
'Name', 'P3D');
    for k = 1:n
        waitbar(k / n, wtsect2);
        % coordinates of the node
        Px = X(k,1);    Py = Y(k,1);    Pz = Z(k,1);
        % conversion of coordinates in cell elements
        Pxc = num2cell(Px);
        Pyc = num2cell(Py);
        Pzc = num2cell(Pz);
        % coupling of coordinates/element name
        for k1 = 1:n
            tfx = isequal(COOR(k1,2),Pxc);
            if tfx == 1
                tfy = isequal(COOR(k1,3),Pyc);
                if tfy == 1
                    tfz = isequal(COOR(k1,4),Pzc);
                    if tfz == 1
                        C=COOR(k1,1) % C = name of the element
                        % containing the node
                    end
                end
            end
        end
        i = 1;
        % First "for" cycle, all connections with C element in first
        % position are finded
        for k2 = 1:m
            tf = isequal(E(k2,1),C);
            if tf == 1
                % if C element is found, I matrix is saved with cell
                % elements
                I(i,1) = E(k2,3);  % Column 1, VELLIQ
                I(i,2) = E(k2,1);  % Column 2, ELEM1 C element
                I(i,3) = E(k2,2);  % Column 3, ELEM2
                % Coordinates of ELEM1, columns 4,5,6 of I
                I(i,4) = Xc(k);
                I(i,5) = Yc(k);
                I(i,6) = Zc(k);
                % "for" cycle for coordinates of ELEM2 of the
```

```
        % connection
        for k3 = 1:n
            tf1 = strncmpi(I(i,3),COOR(k3,1),3);
                if tf1 == 1
                    % Coordinates of ELEM2, columns 7,8,9 of I
                    I(i,7) = COOR(k3,2);
                    I(i,8) = COOR(k3,3);
                    I(i,9) = COOR(k3,4);
                end
        end
        % Number of faces of C element, column 10 of I
        I(i,10) = n_faces(k);
        % Column 18 of I = 1 if C is ELEM1
        I(i,18) = {1};
        i = i+1;
    end
end
% Second "for" cycle, all connections with C element in second
% position are finded
for k2 = 1:m
    tf = isequal(E(k2,2),C);
    if tf == 1
        % if C element is finded, I matrix is saved with cell
        % elements
        I(i,1) = E(k2,3);  % Column 1, VELLIQ
        I(i,2) = E(k2,1);  % Column 2, ELEM1
        I(i,3) = E(k2,2);  % Column 3, ELEM2 C element
        % Coordinates of ELEM2, columns 7,8,9 of I
        I(i,7) = Xc(k);
        I(i,8) = Yc(k);
        I(i,9) = Zc(k);
        % "for" cycle for coordinates of ELEM1 of the
        % connection
        for k3 = 1:n
            tf1 = strncmpi(I(i,2),COOR(k3,1),3);
            if tf1 == 1
               % Coordinates of ELEM1, columns 4,5,6 of I
                    I(i,4) = COOR(k3,2);
                    I(i,5) = COOR(k3,3);
                    I(i,6) = COOR(k3,4);
            end
        end
        % Number of faces of C element, column 10 of I
        I(i,10) = n_faces(k);
        % Column 18 of I = 2 if C is ELEM2
        I(i,18) = {2};
        i = i+1;
    end
end
% Creation of the matrix Id, identical to I but formed by
% numeric elements
Id(:,1) = cell2mat(I(:,1));
Id(:,4:10) = cell2mat(I(:,4:10));
Id(:,18) = cell2mat(I(:,18));

%Individuation of unit vectors perpendicular to the faces of C
% Calculation of the components of the vectors joining the
% node of C with the node of adjacent elements,
% columns 11,12,13 of Id
```

```
        Id(:,11) = Id(:,7)-Id(:,4);
        Id(:,12) = Id(:,8)-Id(:,5);
        Id(:,13) = Id(:,9)-Id(:,6);
        % Norm of the vector, column 14 of Id
        for k4 = 1:size(Id,1)
            Id(k4,14) = norm([Id(k4,11) Id(k4,12) Id(k4,13)]);
        end
        % Components of the unit vector, columns 15,16,17 of Id
        Id(:,15) = Id(:,11)./Id(:,14);
        Id(:,16) = Id(:,12)./Id(:,14);
        Id(:,17) = Id(:,13)./Id(:,14);
        % Darcy's velocities as proposed by Painter
        gamma = Id(:,1); G=Id(:,15:17);
        nG = ((G'*G)^-1)*G';
        vi = nG*gamma;
        % VF contains components of unit vectors of the faces on
        % domain boundaires
        [VF] = ffaces(k, n_faces, faces, Id);
        % Darcy's velocities as proposed by Painter for boundary
        % elements
        bordfaces = size(VF,1);
        % Number of faces on the domain boundaries, column 19 of Id
        Id(:,19) = bordfaces;
        if bordfaces ~= 0
            B = VF(:,1:3);
            % No-flow condition at domain boundaries
            beta = zeros(size(B,1),1);
            % Darcy's velocities as proposed by Painter for boundary
            % elements
            nGB = ((G'*G)^-1)*B'*((B*((G'*G)^-1)*B')^-1);
            vib = vi-nGB*(B*vi-beta);
        end
        % Creation of matrix M, in columns 1,2,3 there are the
        % coordinates of the nodes, in columns 4,5,6 there are the
        % components of the velocity associated to the node, vi if it
        % is a internal node,vib if it is a node on the domain
        % boundaries
        M(k,1:3) = [Px Py Pz];
        if bordfaces == 0
            M(k,4:6) = -vi';
        else
            M(k,4:6) = -vib';
        end
        clear Id I
    end
    % Velocity field is saved in a txt file
    filename = ['velocityfield.txt'];
    fid=fopen(filename,'w');
    fprintf(fid, [ 'X            ' 'Y            ' 'Z            '...
                   'VX           ' 'VY           ' 'VZ           '
                   \n']);
    fprintf(fid, '%.5e %.5e %.5e %.5e %.5e %.5e \n', ...
        [M(:,1) M(:,2) M(:,3) M(:,4) M(:,5) M(:,6)]');
    fclose(fid);
    close(wtsect2);
end

% ######################## END OF SECTION 2 ##########################

%%
```

```
% ############################ SECTION 3 ##############################
% CREATION OF DELAUNAY TESSELLATION

T = delaunayn(M(:,1:3));
% Each row of T is a Delaunay element, indentified with its 4.
% vertices The vertices are identified as indexes of M matrix

% ######################### END OF SECTION 3 ###########################
%%
% ############################ SECTION 4 ##############################
% PATHLINE TRACING

wtsect4 = waitbar(0,'Section 4 - Pathline tracing','Name','P3D');
for n_seed = 1:size(seed)
    % First position of the pathline is found in seed matrix
    XP(1,n_seed) = seed(n_seed,1);
    YP(1,n_seed) = seed(n_seed,2);
    ZP(1,n_seed) = seed(n_seed,3);
    n_it = 2; % Number of iterations
    CDA = 0; % Stop criterion
    while CDA == 0;
        P = [XP(n_it-1,n_seed) YP(n_it-1,n_seed) ZP(n_it-1,n_seed)];
        [ti, bc] = tsearchn(M(:,1:3),T,P);
        % ti = Delaunay's element containing the point P
        % bc = barycentric coordinates of the vertices
        K = ti
        if isnan(ti)
            CDA = 1;
        else
        % Velocities at the vertices
        v1 = M(T(ti,1), 4:6);
        v2 = M(T(ti,2), 4:6);
        v3 = M(T(ti,3), 4:6);
        v4 = M(T(ti,4), 4:6);
        % Velocity in P is the weighted average of velocities at the
        % vertices of Delaunay's element, barycentric coordinates are
        % use as weights
        vx = v1(1)*bc(1) + v2(1)*bc(2) + v3(1)*bc(3) + v4(1)*bc(4);
        vy = v1(2)*bc(1) + v2(2)*bc(2) + v3(2)*bc(3) + v4(2)*bc(4);
        vz = v1(3)*bc(1) + v2(3)*bc(2) + v3(3)*bc(3) + v4(3)*bc(4);
        VX(n_it,n_seed) = vx;
        VY(n_it,n_seed) = vy;
        VZ(n_it,n_seed) = vz;
        % Time step
        dt(n_it,n_seed) = ds_min / (vx^2 + vy^2 + vz^2)^0.5;
        % Position increase
        sx = vx*dt(n_it,n_seed);
        sy = vy*dt(n_it,n_seed);
        sz = vz*dt(n_it,n_seed);
        % New coordinates
        XP(n_it,n_seed) = XP(n_it-1,n_seed) + sx;
        YP(n_it,n_seed) = YP(n_it-1,n_seed) + sy;
        ZP(n_it,n_seed) = ZP(n_it-1,n_seed) + sz;
        % It is started the calculation of a new pathline
        % - if the new position is close to the final position
        % - if it reached the maximum number of iterations
        if abs(XP(n_it,n_seed)-XF) < tolx && ...
           abs(YP(n_it,n_seed)-YF) < toly && ...
```

```
            abs(ZP(n_it,n_seed)-ZF) < tolz
            CDA = 1;
        elseif n_it > n_it_max
            CDA = 1;
        else
            CDA = 0;
        end
        % Time of Fligth
        TOF(n_it,n_seed) = sum(dt(:,n_seed));
        n_it = n_it + 1;
        end
    end
    waitbar(n_seed / size(seed,1), wtsect4);
end
XP(find(XP==0)) = NaN;
YP(find(YP==0)) = NaN;
ZP(find(ZP==0)) = NaN;
close(wtsect4);

% ######################### END OF SECTION 4 ############################
%%
% ############################## SECTION 5 ##############################
% PATHLINES DISPLAY AND SAVE

for k = 1:size(seed,1)
    scatter3(XP(:,k),YP(:,k),ZP(:,k),2,ZP(:,k))
    hold on
    % Each pathline is saved in a txt file
    filename = ['Pathline' num2str(k) '.txt'];
    fid=fopen(filename,'w');
    fprintf(fid, [ 'X[m]      ' 'Y[m]      ' 'Z[m]      ' 'TOF[sec]  '
                   'VX[m/s]      ' 'VY[m/s]      ' 'VZ[m/s]    ''\n']);
    for k4 = 1:size(XP(:,k),1)
        if ~isnan(XP(k4,k))
            k_stop(k) = k4;
        end
    end
    fprintf(fid, '%f %f %f %d %e %e %e \n', ...
            [XP(1:k_stop,k) YP(1:k_stop,k) ZP(1:k_stop,k)
            TOF(1:k_stop,k)...
            VX(1:k_stop,k) VY(1:k_stop,k) VZ(1:k_stop,k)]');
    fclose(fid);
end

for k = 1:size(seed,1)
    text(XP(1,k), YP(1,k), ZP(1,k), num2str(k));
end
title('Pathline display')
axis image
xlabel('X  [m]'); ylabel('Y  [m]'); zlabel('Z  [m]');
grid on
grid minor
cbt = colorbar;
xlabel(cbt,'Z-coordinate  [m]')
hold off
% ######################### END OF SECTION 5 ############################
% ######################### END OF THE CODE #############################
```

importseed.m per P3D

```
%
########################################################################
%NAME    :  importseed.m
%PURPOSE :  for P3D, import data from file specified in input
%AUTHOR  :  Matteo Scapolo
%
%INPUT   :  seed.txt, startRow = 1, endRow = inf
%OUTPUT  :  seed
%
########################################################################
%
function seed = importfile(filename, startRow, endRow)
%% Initialize variables.
if nargin<=2
    startRow = 1;
    endRow = inf;
end

%% Format string for each line of text:
%   column1: double (%f)
%   column2: double (%f)
%   column3: double (%f)
% For more information, see the TEXTSCAN documentation.
formatSpec = '%10f%10f%f%[^\n\r]';

%% Open the text file.
fileID = fopen(filename,'r');

%% Read columns of data according to format string.
% This call is based on the structure of the file used to generate
% this code. If an error occurs for a different file, try regenerating
% the code from the Import Tool.
dataArray = textscan(fileID, formatSpec, endRow(1)-startRow(1)+1,
'Delimiter', '', 'WhiteSpace', '', 'HeaderLines', startRow(1)-1,
'ReturnOnError', false);
for block=2:length(startRow)
    frewind(fileID);
    dataArrayBlock = textscan(fileID, formatSpec, endRow(block)-
startRow(block)+1, 'Delimiter', '', 'WhiteSpace', '', 'HeaderLines',
startRow(block)-1, 'ReturnOnError', false);
    for col=1:length(dataArray)
        dataArray{col} = [dataArray{col};dataArrayBlock{col}];
    end
end

%% Close the text file.
fclose(fileID);

%% Post processing for unimportable data.
% No unimportable data rules were applied during the import, so no
% post processing code is included. To generate code which works for
% unimportable data, select unimportable cells in a file and
% regenerate the script.
%% Create output variable
seed = [dataArray{1:end-1}];
```

importvelocityfield.m per P3D

```
%
########################################################################
%NAME    :  importvelocityfield.m
%PURPOSE :  for P3D, import data from file specified in input
%AUTHOR  :  Matteo Scapolo
%
%INPUT   :  velocityfield.txt.txt, startRow = 1, endRow = inf
%OUTPUT  :  velocityfield
%
########################################################################
%
%
function velocityfield = importfile(filename, startRow, endRow)
%% Initialize variables.
delimiter = ' ';
if nargin<=2
    startRow = 2;
    endRow = inf;
end

%% Format string for each line of text:
%   column1: double (%f)
%   column2: double (%f)
%   column3: double (%f)
%   column4: double (%f)
%   column5: double (%f)
%   column6: double (%f)
% For more information, see the TEXTSCAN documentation.
formatSpec = '%f%f%f%f%f%f%[^\n\r]';

%% Open the text file.
fileID = fopen(filename,'r');

%% Read columns of data according to format string.
% This call is based on the structure of the file used to generate
% this code. If an error occurs for a different file, try regenerating
% the code from the Import Tool.
dataArray = textscan(fileID, formatSpec, endRow(1)-startRow(1)+1,
'Delimiter', delimiter, 'MultipleDelimsAsOne', true, 'HeaderLines',
startRow(1)-1, 'ReturnOnError', false);
for block=2:length(startRow)
    frewind(fileID);
    dataArrayBlock = textscan(fileID, formatSpec, endRow(block)-
startRow(block)+1, 'Delimiter', delimiter, 'MultipleDelimsAsOne',
true, 'HeaderLines', startRow(block)-1, 'ReturnOnError', false);
    for col=1:length(dataArray)
        dataArray{col} = [dataArray{col};dataArrayBlock{col}];
    end
end

%% Close the text file.
fclose(fileID);

%% Post processing for unimportable data.
% No unimportable data rules were applied during the import, so no
```

```
% post processing code is included. To generate code which works for
% unimportable data, select unimportable cells in a file and e the
% regenerate the script.

%% Create output variable
velocityfield = [dataArray{1:end-1}];
```

importelements.m per P3D

```
%
########################################################################
%NAME    :  importelements.m
%PURPOSE :  for P3D, import data from file specified in input
%AUTHOR  :  Matteo Scapolo
%
%INPUT   :  elements.txt, startRow = 1, endRow = Inf
%OUTPUT  :  ELEMENTS,X,Y,Z
%
########################################################################
%
%
function [ELEMENTS,X,Y,Z] = importfile(filename, startRow, endRow)
%% Initialize variables.
if nargin<=2
    startRow = 1;
    endRow = inf;
end

%% Read columns of data as strings:
% For more information, see the TEXTSCAN documentation.
formatSpec = '%5s%*15*s%*10*s%*10*s%*10s%10s%10s%s%[^\n\r]';

%% Open the text file.
fileID = fopen(filename,'r');

%% Read columns of data according to format string.
% This call is based on the structure of the file used to generate
% this code. If an error occurs for a different file, try regenerating
% the code from the Import Tool.
dataArray = textscan(fileID, formatSpec, endRow(1)-startRow(1)+1,
'Delimiter', '', 'WhiteSpace', '', 'HeaderLines', startRow(1)-1,
'ReturnOnError', false);
for block=2:length(startRow)
    frewind(fileID);
    dataArrayBlock = textscan(fileID, formatSpec, endRow(block)-
startRow(block)+1, 'Delimiter', '', 'WhiteSpace', '', 'HeaderLines',
startRow(block)-1, 'ReturnOnError', false);
    for col=1:length(dataArray)
        dataArray{col} = [dataArray{col};dataArrayBlock{col}];
    end
end

%% Remove white space around all cell columns.
dataArray{1} = strtrim(dataArray{1});

%% Close the text file.
fclose(fileID);

%% Convert the contents of columns containing numeric strings to
% numbers. Replace non-numeric strings with NaN.
raw = repmat({''},length(dataArray{1}),length(dataArray)-1);
for col=1:length(dataArray)-1
    raw(1:length(dataArray{col}),col) = dataArray{col};
end
```

```
numericData = NaN(size(dataArray{1},1),size(dataArray,2));

for col=[2,3,4]
    % Converts strings in the input cell array to numbers. Replaced
    % non-numeric strings with NaN.
    rawData = dataArray{col};
    for row=1:size(rawData, 1);
        % Create a regular expression to detect and remove non-numeric
        % prefixes and suffixes.
        regexstr = '(?<prefix>.*?)(?<numbers>([-
]*(\d+[\,]*)+[\.]{0,1}\d*[eEdD]{0,1}[-+]*\d*[i]{0,1})|([-
]*(\d+[\,]*)*[\.]{1,1}\d+[eEdD]{0,1}[-+]*\d*[i]{0,1}))(?<suffix>.*)';
        try
            result = regexp(rawData{row}, regexstr, 'names');
            numbers = result.numbers;

            % Detected commas in non-thousand locations.
            invalidThousandsSeparator = false;
            if any(numbers==',');
                thousandsRegExp = '^\d+?(\,\d{3})*\.{0,1}\d*$';
                if isempty(regexp(numbers, thousandsRegExp, 'once'));
                    numbers = NaN;
                    invalidThousandsSeparator = true;
                end
            end
            % Convert numeric strings to numbers.
            if ~invalidThousandsSeparator;
                numbers = textscan(strrep(numbers, ',', ''), '%f');
                numericData(row, col) = numbers{1};
                raw{row, col} = numbers{1};
            end
        catch me
        end
    end
end


%% Split data into numeric and cell columns.
rawNumericColumns = raw(:, [2,3,4]);
rawCellColumns = raw(:, 1);


%% Exclude rows with non-numeric cells
I = ~all(cellfun(@(x) (isnumeric(x) || islogical(x)) &&
~isnan(x),rawNumericColumns),2); % Find rows with non-numeric cells
rawNumericColumns(I,:) = [];
rawCellColumns(I,:) = [];

%% Allocate imported array to column variable names
ELEMENTS = rawCellColumns(:, 1);
X = cell2mat(rawNumericColumns(:, 1));
Y = cell2mat(rawNumericColumns(:, 2));
Z = cell2mat(rawNumericColumns(:, 3));
```

importvelocities.m per P3D

```
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%NAME    :  importvelocities.m
%PURPOSE :  for P3D, import data from file specified in input
%AUTHOR  :  Matteo Scapolo
%
%INPUT   :  velocities.txt, startRow = 1, endRow = inf
%OUTPUT  :  ELEM1,ELEM2,VELLIQ
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%
function [ELEM1,ELEM2,VELLIQ] = importfile(filename, startRow, endRow)
%% Initialize variables.
if nargin<=2
    startRow = 1;
    endRow = inf;
end

%% Read columns of data as strings:
% For more information, see the TEXTSCAN documentation.
formatSpec = '%5s%5s%s%[^\n\r]';

%% Open the text file.
fileID = fopen(filename,'r');

%% Read columns of data according to format string.
% This call is based on the structure of the file used to generate
% this code. If an error occurs for a different file, try regenerating
% the code from the Import Tool.
dataArray = textscan(fileID, formatSpec, endRow(1)-startRow(1)+1,
'Delimiter', '', 'WhiteSpace', '', 'HeaderLines', startRow(1)-1,
'ReturnOnError', false);
for block=2:length(startRow)
    frewind(fileID);
    dataArrayBlock = textscan(fileID, formatSpec, endRow(block)-
startRow(block)+1, 'Delimiter', '', 'WhiteSpace', '', 'HeaderLines',
startRow(block)-1, 'ReturnOnError', false);
    for col=1:length(dataArray)
        dataArray{col} = [dataArray{col};dataArrayBlock{col}];
    end
end

%% Remove white space around all cell columns.
dataArray{1} = strtrim(dataArray{1});
dataArray{2} = strtrim(dataArray{2});

%% Close the text file.
fclose(fileID);

%% Convert the contents of columns containing numeric strings to
% numbers. Replace non-numeric strings with NaN.
raw = repmat({''},length(dataArray{1}),length(dataArray)-1);
for col=1:length(dataArray)-1
    raw(1:length(dataArray{col}),col) = dataArray{col};
```

```
end
numericData = NaN(size(dataArray{1},1),size(dataArray,2));

% Converts strings in the input cell array to numbers. Replaced non-
% numeric strings with NaN.
rawData = dataArray{3};
for row=1:size(rawData, 1);
    % Create a regular expression to detect and remove non-numeric
    % prefixes and suffixes.
    regexstr = '(?<prefix>.*?)(?<numbers>([-
]*(\d+[\,]*)+[\.]{0,1}\d*[eEdD]{0,1}[-+]*\d*[i]{0,1})|([-
]*(\d+[\,]*)*[\.]{1,1}\d+[eEdD]{0,1}[-+]*\d*[i]{0,1}))(?<suffix>.*)';
    try
        result = regexp(rawData{row}, regexstr, 'names');
        numbers = result.numbers;

        % Detected commas in non-thousand locations.
        invalidThousandsSeparator = false;
        if any(numbers==',');
            thousandsRegExp = '^\d+?(\,\d{3})*\.{0,1}\d*$';
            if isempty(regexp(numbers, thousandsRegExp, 'once'));
                numbers = NaN;
                invalidThousandsSeparator = true;
            end
        end
        % Convert numeric strings to numbers.
        if ~invalidThousandsSeparator;
            numbers = textscan(strrep(numbers, ',', ''), '%f');
            numericData(row, 3) = numbers{1};
            raw{row, 3} = numbers{1};
        end
    catch me
    end
end


%% Split data into numeric and cell columns.
rawNumericColumns = raw(:, 3);
rawCellColumns = raw(:, [1,2]);


%% Exclude rows with non-numeric cells
I = ~all(cellfun(@(x) (isnumeric(x) || islogical(x)) &&
~isnan(x),rawNumericColumns),2); % Find rows with non-numeric cells
rawNumericColumns(I,:) = [];
rawCellColumns(I,:) = [];

%% Allocate imported array to column variable names
ELEM1 = rawCellColumns(:, 1);
ELEM2 = rawCellColumns(:, 2);
VELLIQ = cell2mat(rawNumericColumns(:, 1));
```

importfaces.m per P3D

```
%
########################################################################
%NAME    :  importfaces.m
%PURPOSE :  for P3D, import data from file specified in input
%AUTHOR  :  Matteo Scapolo
%
%INPUT   :  faces.txt, startRow = 1, endRow = inf, n
%OUTPUT  :  faces, n_faces
%
########################################################################
%
%
function [faces, n_faces] = importfile(filename, startRow, endRow, n)
%% Initialize variables.
delimiter = ' ';
if nargin<=2
    startRow = 1;
    endRow = inf;
end

%% Format string for each line of text:
%   column1: double (%f)
% ...
%   column5: double (%f)
%   column6: text (%s)
% ...
%   column42: text (%s)
% For more information, see the TEXTSCAN documentation.
formatSpec =
'%f%f%f%f%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%
s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%
s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%
s%[^\n\r]';

%% Open the text file.
fileID = fopen(filename,'r');

%% Read columns of data according to format string.
% This call is based on the structure of the file used to generate
% this code. If an error occurs for a different file, try regenerating
% the code from the Import Tool.
dataArray = textscan(fileID, formatSpec, endRow(1)-startRow(1)+1,
'Delimiter', delimiter, 'MultipleDelimsAsOne', true, 'HeaderLines',
startRow(1)-1, 'ReturnOnError', false);
for block=2:length(startRow)
    frewind(fileID);
    dataArrayBlock = textscan(fileID, formatSpec, endRow(block)-
startRow(block)+1, 'Delimiter', delimiter, 'MultipleDelimsAsOne',
true, 'HeaderLines', startRow(block)-1, 'ReturnOnError', false);
    for col=1:length(dataArray)
        dataArray{col} = [dataArray{col};dataArrayBlock{col}];
    end
end

%% Close the text file.
```

```
fclose(fileID);

%% Post processing for unimportable data.
% No unimportable data rules were applied during the import, so no
% post processing code is included. To generate code which works for
% unimportable data, select unimportable cells in a file and the
% regenerate script.

%% Create output variable
dataArray([1, 2, 3, 4, 5]) = cellfun(@(x) num2cell(x), dataArray([1,
2, 3, 4, 5]), 'UniformOutput', false);
faces = [dataArray{1:end-1}];

i=1; CStr = textread('faces.txt', '%s', 'delimiter', '\n');
for k=1:n
    String=CStr{k};
    Strings=regexp(String, '(?<=\) )\d+(?= \()', 'match');
    n_faces(i,1)=num2cell(str2double(cell2mat(Strings)));
    i=i+1;
end
```

ffaces.m per P3D

```
%
########################################################################
###
%NAME    :  ffaces.m
%PURPOSE :  for P3D, creation of the matrix VF, containing the
%           components of unit vectors normal to the faces on domain
%           boundaries
%AUTHOR  :  Matteo Scapolo
%
%INPUT   :  k, n_faces, faces, Id
%OUTPUT  :  VF
%
########################################################################
%
%
function [VF] = ffaces(k, n_faces, faces, Id)
% Import of the number of faces from tough2viewer.dat
% Individuation of the column containing the number of faces
    for k5 = 1:size(faces,2)
        tf = strncmpi(num2str(n_faces{k}),faces{k,k5},3);
        if tf == 1
            column_faces = k5;
        end
    end
    % Individuation of all unit vectors from tough2viewer.dat
    for k6 = 1:cell2mat(n_faces(k,1))
        string_vector(k6,:) = faces(k,column_faces +
cell2mat(n_faces(k,1)) + k6);
        VF(k6,:) = sscanf(string_vector{k6},'(%f,%f,%f)',[1 Inf]);
    end
    check_matrix = VF;
    % Elimination from VF of unit vectors shared by Id and
    % tough2viewer.dat Unit vectors remaining are perpendicolar to
    % faces on the domain boundaries
    for cc = 1:size(Id,1)
        if Id(cc,18) == 1
            VF(abs(VF(:,1) - Id(cc,15)) < 0.1...
             & abs(VF(:,2) - Id(cc,16)) < 0.1...
             & abs(VF(:,3) - Id(cc,17)) < 0.1,:) = [];
        else
            VF(abs(VF(:,1) + Id(cc,15)) < 0.1...
             & abs(VF(:,2) + Id(cc,16)) < 0.1...
             & abs(VF(:,3) + Id(cc,17)) < 0.1,:) = [];
        end
    end
end
```

Appendice B

P2D

```
%
######################################################################
%NAME     :  P2D.m
%PURPOSE  :  Pathline tracing on fully unstructured Voronoi 2D grid
%AUTHOR   :  Matteo Scapolo
%
%INPUT 1  :  seed.txt,velocityfield.txt
%INPUT 2  :  seed.txt,elements.txt,velocities.txt,faces.txt
%OUTPUT   :  Pathline display, Pathline1.txt, Pathline2.txt, etc.
%
######################################################################
clear all
clc
close all
%%
% ##################### SECTION 1 ###################################
% DEFINITION OF THE PARAMETERS BY THE USER

% Min. displacement
ds_min = 1; % meters

% Max. iterations
n_it_max = 10^5;

% Final position
XF = 500;     YF = 500; %meters

% Final position tolerances
tolx = 5;     toly = 5; %meters

% Velocity field is already defined? 0 for NO, 1 for YES
VField = 0;

% Pathline seeds import
[seed] = importseed('seed.txt', 1, inf)

% ######################### END OF SECTION 1 ########################
%%
% ################### SECTION 2 #####################################
% DEFINITION OF THE VELOCITY FIELD

if VField == 1; % Velocity field is already defined
   M = importvelocityfield('velocityfield.txt', 2, inf);
else % VField == 0;  % Velocity field is NOT defined

    % IMPORT DATA
    [ELEMENTS,X,Y,~] = importelements('elements.txt',1,inf);
    % ELEMENTS = Elements names
```

```
    % X,Y      = Coordinates of the centers of the elements
    [ELEM1,ELEM2,VELLIQ] = importvelocities('velocities.txt',1,inf);
    % ELEM1  = Fist element of the connection
    % ELEM2  = Second element of the connection
    % VELLIQ = Velocity of the fluid at the interface
    n = size(ELEMENTS,1); %number of elements
    m = size(ELEM1,1); %number of connections
    [faces, n_faces] = importfaces('faces.txt', 1, inf,n);
    % faces   = Passing variable
    % n_faces = Number of total faces of the element

    % Conversion of numerical vectors in alphanumeric vectors
    VELLIQ_cell = num2cell(VELLIQ);
    Xc = num2cell(X);    Yc = num2cell(Y);
    E = [ELEM1, ELEM2, VELLIQ_cell]; % alphanumeric matrix
    COOR = [ELEMENTS,Xc,Yc]; % alphanumeric matrix

    % VELOCITIES ASSIGNMENT TO VORONOI NODES
wtsect2 = waitbar(0,'Section 2 - Definition of the velocity field',
'Name', 'P2D');
    for k = 1:n
        waitbar(k / n, wtsect2);
        % coordinates of the node
        Px = X(k,1);    Py = Y(k,1);
        % conversion of coordinates in cell elements
        Pxc = num2cell(Px);    Pyc = num2cell(Py);
        % coupling of coordinates/element name
        for k1 = 1:n
            tfx = isequal(COOR(k1,2),Pxc);
            if tfx == 1
                tfy = isequal(COOR(k1,3),Pyc);
                if tfy == 1
                   C=COOR(k1,1) % C = name of the element containing
                   % the node
                end
            end
        end
        i = 1;
        % First "for" cycle, all connections with C element in first
        % position are finded
        for k2 = 1:m
            tf = isequal(E(k2,1),C);
            if tf == 1
                % if C element is finded, I matrix is saved with cell
                % elements
                I(i,1) = E(k2,3);  % Column 1, VELLIQ
                I(i,2) = E(k2,1);  % Column 2, ELEM1 C element
                I(i,3) = E(k2,2);  % Column 3, ELEM2
                % Coordinates of ELEM1, columns 4,5 of I
                I(i,4) = Xc(k);
                I(i,5) = Yc(k);
                % "for" cycle for coordinates of ELEM2 of the
                % connection
                for k3 = 1:n
                    tf1 = strncmpi(I(i,3),COOR(k3,1),3);
                        if tf1 == 1
                            % Coordinates of ELEM2, columns 6,7 of I
                            I(i,6) = COOR(k3,2);
```

```
                    I(i,7) = COOR(k3,3);
                end
            end
            % Number of faces of C element, column 8 of I
            I(i,8) = n_faces(k);
            % Column 14 of I = 1 if C is ELEM1
            I(i,14) = {1};
            i = i+1;
        end
    end
    % Second "for" cycle, all connections with C element in second
    % position are finded
    for k2 = 1:m
        tf = isequal(E(k2,2),C);
        if tf == 1
            % if C element is finded, I matrix is saved with cell
            % elements
            I(i,1) = E(k2,3);  % Column 1, VELLIQ
            I(i,2) = E(k2,1);  % Column 2, ELEM1
            I(i,3) = E(k2,2);  % Column 3, ELEM2 C element
            % Coordinates of ELEM2, columns 6,7 of I
            I(i,6) = Xc(k);
            I(i,7) = Yc(k);
            % "for" cycle for coordinates of ELEM1 of the
            % connection
            for k3 = 1:n
                tf1 = strncmpi(I(i,2),COOR(k3,1),3);
                if tf1 == 1
                   % Coordinates of ELEM1, columns 4,5 of I
                        I(i,4) = COOR(k3,2);
                        I(i,5) = COOR(k3,3);
                end
            end
            % Number of faces of C element, column 8 of I
            I(i,8) = n_faces(k);
            % Column 14 of I = 2 if C is ELEM2
            I(i,14) = {2};
            i = i+1;
        end
    end
    % Creation of the matrix Id, identical to I but formed by
    % numeric elements
    Id(:,1) = cell2mat(I(:,1));
    Id(:,4:8) = cell2mat(I(:,4:8));
    Id(:,14) = cell2mat(I(:,14));

    %Individuation of unit vectors perpendicular to the faces of C
    % Calculation of the components of the vectors joining the
    % node of C with the node of adjacent elements, columns 9,10
    % of Id
    Id(:,9) = Id(:,6)-Id(:,4);
    Id(:,10) = Id(:,7)-Id(:,5);
    % Norm of the vector, column 14 of Id
    for k4 = 1:size(Id,1)
        Id(k4,11) = norm([Id(k4,9) Id(k4,10)]);
    end
    % Components of the unit vector, columns 12,13 of Id
    Id(:,12) = Id(:,9)./Id(:,11);
    Id(:,13) = Id(:,10)./Id(:,11);
```

```
        % Darcy's velocities as proposed by Painter
        gamma = Id(:,1); G=Id(:,12:13);
        nG = ((G'*G)^-1)*G';
        vi = nG*gamma;
        % VF contains components of unit vectors of the faces on
        % domain boundaires
        [VF] = ffaces(k, n_faces, faces, Id);
        % Darcy's velocities as proposed by Painter for boundary
        % elements
        bordfaces = size(VF,1);
        % Number of faces on the domain boundaries, column 15 of Id
        Id(:,15) = bordfaces;
        if bordfaces ~= 0
            B = VF(:,1:2);
            % No-flow condition at domain boundaries
            beta = zeros(size(B,1),1);
            % Darcy's velocities as proposed by Painter for boundary
            % elements
            nGB = ((G'*G)^-1)*B'*((B*((G'*G)^-1)*B')^-1);
            vib = vi-nGB*(B*vi-beta);
        end
        % Creation of matrix M, in columns 1,2 there are the
        % coordinates of the nodes, in columns 3,4 there are the
        % components of the velocity associated to the node, vi if it
        % is a internal node,vib if it is a node on the domain
        % boundaries
        M(k,1:2) = [Px Py];
        if bordfaces == 0
            M(k,3:4) = -vi';
        else
            M(k,3:4) = -vib';
        end
        clear Id I
    end
    % Velocity field is saved in a txt file
    filename = ['velocityfield.txt'];
    fid=fopen(filename,'w');
    fprintf(fid, [ 'X            ' 'Y           '...
                   'VX           ' 'VY        \n']);
    fprintf(fid, '%.5e %.5e %.5e %.5e \n', ...
        [M(:,1) M(:,2) M(:,3) M(:,4)]');
    fclose(fid);
    close(wtsect2);
end

% ######################### END OF SECTION 2 ############################
%%
% ###################### SECTION 3 ######################################
% CREATION OF DELAUNAY TESSELLATION

T = delaunayn(M(:,1:2));
% Each row of T is a Delaunay element, indentified with its 4
% vertices. The vertices are identified as indexes of M matrix

% ######################### END OF SECTION 3 ############################
```

```
%%
% ############################ SECTION 4 ##############################
% PATHLINE TRACING

wtsect4 = waitbar(0,'Section 4 - Pathline tracing','Name','P3D');
for n_seed = 1:size(seed)
    waitbar(n_seed / size(seed,1), wtsect4);
    % First position of the pathline is found in seed matrix
    XP(1,n_seed) = seed(n_seed,1);
    YP(1,n_seed) = seed(n_seed,2);
    n_it = 2; % Number of iterations
    CDA = 0; % Stop criterion
    while CDA == 0;
        P = [XP(n_it-1,n_seed) YP(n_it-1,n_seed)];
        [ti, bc] = tsearchn(M(:,1:2),T,P);
        % ti = Delaunay's element containing the point P
        % bc = barycentric coordinates of the vertices
        K = ti
        if isnan(ti)
            CDA = 1;
        else
        % Velocities at the vertices
        v1 = M(T(ti,1), 3:4);
        v2 = M(T(ti,2), 3:4);
        v3 = M(T(ti,3), 3:4);
        % Velocity in P is the weighted average of velocities at the
        % vertices of Delaunay's element, barycentric coordinates are
        % use as weights
        vx = v1(1)*bc(1) + v2(1)*bc(2) + v3(1)*bc(3);
        vy = v1(2)*bc(1) + v2(2)*bc(2) + v3(2)*bc(3);
        VX(n_it,n_seed) = vx;
        VY(n_it,n_seed) = vy;
        % Time step
        dt(n_it,n_seed) = ds_min / (vx^2 + vy^2)^0.5;
        % Position increase
        sx = vx*dt(n_it,n_seed);
        sy = vy*dt(n_it,n_seed);
        % New coordinates
        XP(n_it,n_seed) = XP(n_it-1,n_seed) + sx;
        YP(n_it,n_seed) = YP(n_it-1,n_seed) + sy;
        % It is started the calculation of a new pathline
        % if the new position is close to the final position
        % if it reached the maximum number of iterations
        if abs(XP(n_it,n_seed)-XF) < tolx && ...
           abs(YP(n_it,n_seed)-YF) < toly
           CDA = 1;
        elseif n_it > n_it_max
            CDA = 1;
        else
            CDA = 0;
        end
        % Time of Fligth
        TOF(n_it,n_seed) = sum(dt(:,n_seed));
        n_it = n_it + 1;
        end
    end
end
XP(find(XP==0)) = NaN;
YP(find(YP==0)) = NaN;
```

```
close(wtsect4);

% ######################### END OF SECTION 4 ############################
%%
% ############################ SECTION 5 ################################
% PATHLINES DISPLAY AND SAVE

for k = 1:size(seed,1)
    % Each pathline is saved in a txt file
    filename = ['Pathline' num2str(k) '.txt'];
    fid=fopen(filename,'w');
    fprintf(fid, [ 'X          ' 'Y          ' 'VX[m/s]      '...
                   'VY[m/s]      ' 'TOF        ' '\n']);
    for k4 = 1:size(XP(:,k),1)
        if ~isnan(XP(k4,k))
            k_stop(k) = k4;
        end
    end
    fprintf(fid, '%f %f %d %e %e \n', [XP(1:k_stop,k) YP(1:k_stop,k)
        TOF(1:k_stop,k) VX(1:k_stop,k) VY(1:k_stop,k)]');
    fclose(fid);
end

plot(XP,YP,'MarkerSize',2,'LineWidth',2)
title('Pathline display')
axis image
xlabel('X  [m]'); ylabel('Y  [m]');
for k = 1:size(seed,1)
    text(XP(1,k), YP(1,k), num2str(k));
end
grid on
grid minor

% ######################### END OF SECTION 5 ############################
% ######################### END OF THE CODE #############################
```

importvelocityfield.m per P2D

```
%
########################################################################
%NAME    :  importvelocityfield.m
%PURPOSE :  for P2D, import data from file specified in input
%AUTHOR  :  Matteo Scapolo
%
%INPUT   :  velocityfield.txt.txt, startRow = 1, endRow = inf
%OUTPUT  :  velocityfield
%
########################################################################
%
%
function velocityfield = importfile(filename, startRow, endRow)
%% Initialize variables.
delimiter = ' ';
if nargin<=2
    startRow = 1;
    endRow = inf;
end

%% Format string for each line of text:
%   column1: double (%f)
%   column2: double (%f)
%   column3: double (%f)
%   column4: double (%f)
% For more information, see the TEXTSCAN documentation.
formatSpec = '%f%f%f%f%[^\n\r]';

%% Open the text file.
fileID = fopen(filename,'r');

%% Read columns of data according to format string.
% This call is based on the structure of the file used to generate
% this code. If an error occurs for a different file, try regenerating
% the code from the Import Tool.
dataArray = textscan(fileID, formatSpec, endRow(1)-startRow(1)+1,
'Delimiter', delimiter, 'MultipleDelimsAsOne', true, 'HeaderLines',
startRow(1)-1, 'ReturnOnError', false);
for block=2:length(startRow)
    frewind(fileID);
    dataArrayBlock = textscan(fileID, formatSpec, endRow(block)-
startRow(block)+1, 'Delimiter', delimiter, 'MultipleDelimsAsOne',
true, 'HeaderLines', startRow(block)-1, 'ReturnOnError', false);
    for col=1:length(dataArray)
        dataArray{col} = [dataArray{col};dataArrayBlock{col}];
    end
end

%% Close the text file.
fclose(fileID);

%% Post processing for unimportable data.
% No unimportable data rules were applied during the import, so no
% post processing code is included. To generate code which works for
```

```
% unimportable data, select unimportable cells in a file and
% regenerate the script.

%% Create output variable
velocityfield = [dataArray{1:end-1}];
```

importseed.m per P2D

```
%
########################################################################
%NAME    :  importseed.m
%PURPOSE :  for P2D, import data from file specified in input
%AUTHOR  :  Matteo Scapolo
%
%INPUT   :  seed.txt, startRow = 1, endRow = inf
%OUTPUT  :  seed
%
########################################################################
%
%
function seed = importfile(filename, startRow, endRow)
%% Initialize variables.
if nargin<=2
    startRow = 1;
    endRow = inf;
end

%% Read columns of data as strings:
% For more information, see the TEXTSCAN documentation.
formatSpec = '%10s%s%[^\n\r]';

%% Open the text file.
fileID = fopen(filename,'r');

%% Read columns of data according to format string.
% This call is based on the structure of the file used to generate
% this code. If an error occurs for a different file, try regenerating
% the code from the Import Tool.
dataArray = textscan(fileID, formatSpec, endRow(1)-startRow(1)+1,
'Delimiter', '', 'WhiteSpace', '', 'HeaderLines', startRow(1)-1,
'ReturnOnError', false);
for block=2:length(startRow)
    frewind(fileID);
    dataArrayBlock = textscan(fileID, formatSpec, endRow(block)-
startRow(block)+1, 'Delimiter', '', 'WhiteSpace', '', 'HeaderLines',
startRow(block)-1, 'ReturnOnError', false);
    for col=1:length(dataArray)
        dataArray{col} = [dataArray{col};dataArrayBlock{col}];
    end
end

%% Close the text file.
fclose(fileID);

%% Convert the contents of columns containing numeric strings to
% numbers. Replace non-numeric strings with NaN.
raw = repmat({''},length(dataArray{1}),length(dataArray)-1);
for col=1:length(dataArray)-1
    raw(1:length(dataArray{col}),col) = dataArray{col};
end
numericData = NaN(size(dataArray{1},1),size(dataArray,2));

for col=[1,2]
```

```
    % Converts strings in the input cell array to numbers. Replaced
    % non-numeric strings with NaN.
    rawData = dataArray{col};
    for row=1:size(rawData, 1);
        % Create a regular expression to detect and remove non-numeric
        % prefixes and suffixes.
        regexstr = '(?<prefix>.*?)(?<numbers>([-
]*(\d+[\,]*)+[\.]{0,1}\d*[eEdD]{0,1}[-+]*\d*[i]{0,1})|([-
]*(\d+[\,]*)*[\.]{1,1}\d+[eEdD]{0,1}[-+]*\d*[i]{0,1}))(?<suffix>.*)';
        try
            result = regexp(rawData{row}, regexstr, 'names');
            numbers = result.numbers;

            % Detected commas in non-thousand locations.
            invalidThousandsSeparator = false;
            if any(numbers==',');
                thousandsRegExp = '^\d+?(\,\d{3})*\.{0,1}\d*$';
                if isempty(regexp(numbers, thousandsRegExp, 'once'));
                    numbers = NaN;
                    invalidThousandsSeparator = true;
                end
            end
            % Convert numeric strings to numbers.
            if ~invalidThousandsSeparator;
                numbers = textscan(strrep(numbers, ',', ''), '%f');
                numericData(row, col) = numbers{1};
                raw{row, col} = numbers{1};
            end
        catch me
        end
    end
end


%% Create output variable
seed = cell2mat(raw);
```

importelements.m per P2D

```
%
########################################################################
%NAME    :  importelements.m
%PURPOSE :  for P2D, import data from file specified in input
%AUTHOR  :  Matteo Scapolo
%
%INPUT   :  elements.txt, startRow = 1, endRow = Inf
%OUTPUT  :  ELEMENTS,X,Y,~
%
########################################################################
%
%
function [ELEMENTS,X,Y,Z] = importfile(filename, startRow, endRow)
%% Initialize variables.
if nargin<=2
    startRow = 1;
    endRow = inf;
end

%% Read columns of data as strings:
% For more information, see the TEXTSCAN documentation.
formatSpec = '%5s%*15*s%*10*s%*10*s%*10s%10s%10s%s%[^\n\r]';

%% Open the text file.
fileID = fopen(filename,'r');

%% Read columns of data according to format string.
% This call is based on the structure of the file used to generate
% this code. If an error occurs for a different file, try regenerating
% the code from the Import Tool.
dataArray = textscan(fileID, formatSpec, endRow(1)-startRow(1)+1,
'Delimiter', '', 'WhiteSpace', '', 'HeaderLines', startRow(1)-1,
'ReturnOnError', false);
for block=2:length(startRow)
    frewind(fileID);
    dataArrayBlock = textscan(fileID, formatSpec, endRow(block)-
startRow(block)+1, 'Delimiter', '', 'WhiteSpace', '', 'HeaderLines',
startRow(block)-1, 'ReturnOnError', false);
    for col=1:length(dataArray)
        dataArray{col} = [dataArray{col};dataArrayBlock{col}];
    end
end

%% Remove white space around all cell columns.
dataArray{1} = strtrim(dataArray{1});

%% Close the text file.
fclose(fileID);

%% Convert the contents of columns containing numeric strings to
% numbers. Replace non-numeric strings with NaN.
raw = repmat({''},length(dataArray{1}),length(dataArray)-1);
for col=1:length(dataArray)-1
    raw(1:length(dataArray{col}),col) = dataArray{col};
end
```

```
numericData = NaN(size(dataArray{1},1),size(dataArray,2));

for col=[2,3,4]
    % Converts strings in the input cell array to numbers. Replaced
    % non-numeric strings with NaN.
    rawData = dataArray{col};
    for row=1:size(rawData, 1);
        % Create a regular expression to detect and remove non-numeric
        % prefixes and suffixes.
        regexstr = '(?<prefix>.*?)(?<numbers>([-
]*(\d+[\,]*)+[\.]{0,1}\d*[eEdD]{0,1}[-+]*\d*[i]{0,1})|([-
]*(\d+[\,]*)*[\.]{1,1}\d+[eEdD]{0,1}[-+]*\d*[i]{0,1}))(?<suffix>.*)';
        try
            result = regexp(rawData{row}, regexstr, 'names');
            numbers = result.numbers;

            % Detected commas in non-thousand locations.
            invalidThousandsSeparator = false;
            if any(numbers==',');
                thousandsRegExp = '^\d+?(\,\d{3})*\.{0,1}\d*$';
                if isempty(regexp(numbers, thousandsRegExp, 'once'));
                    numbers = NaN;
                    invalidThousandsSeparator = true;
                end
            end
            % Convert numeric strings to numbers.
            if ~invalidThousandsSeparator;
                numbers = textscan(strrep(numbers, ',', ''), '%f');
                numericData(row, col) = numbers{1};
                raw{row, col} = numbers{1};
            end
        catch me
        end
    end
end


%% Split data into numeric and cell columns.
rawNumericColumns = raw(:, [2,3,4]);
rawCellColumns = raw(:, 1);


%% Exclude rows with non-numeric cells
I = ~all(cellfun(@(x) (isnumeric(x) || islogical(x)) &&
~isnan(x),rawNumericColumns),2); % Find rows with non-numeric cells
rawNumericColumns(I,:) = [];
rawCellColumns(I,:) = [];

%% Allocate imported array to column variable names
ELEMENTS = rawCellColumns(:, 1);
X = cell2mat(rawNumericColumns(:, 1));
Y = cell2mat(rawNumericColumns(:, 2));
Z = cell2mat(rawNumericColumns(:, 3));
```

importvelocities.m per P2D

```
%
##########################################################################
%NAME    :  importvelocities.m
%PURPOSE :  for P2D, import data from file specified in input
%AUTHOR  :  Matteo Scapolo
%
%INPUT   :  velocities.txt, startRow = 1, endRow = inf
%OUTPUT  :  ELEM1,ELEM2,VELLIQ
%
##########################################################################
%
%
function [ELEM1,ELEM2,VELLIQ] = importfile(filename, startRow, endRow)
%% Initialize variables.
if nargin<=2
    startRow = 1;
    endRow = inf;
end

%% Read columns of data as strings:
% For more information, see the TEXTSCAN documentation.
formatSpec = '%5s%5s%s%[^\n\r]';

%% Open the text file.
fileID = fopen(filename,'r');

%% Read columns of data according to format string.
% This call is based on the structure of the file used to generate
% this code. If an error occurs for a different file, try regenerating
% the code from the Import Tool.
dataArray = textscan(fileID, formatSpec, endRow(1)-startRow(1)+1,
'Delimiter', '', 'WhiteSpace', '', 'HeaderLines', startRow(1)-1,
'ReturnOnError', false);
for block=2:length(startRow)
    frewind(fileID);
    dataArrayBlock = textscan(fileID, formatSpec, endRow(block)-
startRow(block)+1, 'Delimiter', '', 'WhiteSpace', '', 'HeaderLines',
startRow(block)-1, 'ReturnOnError', false);
    for col=1:length(dataArray)
        dataArray{col} = [dataArray{col};dataArrayBlock{col}];
    end
end

%% Remove white space around all cell columns.
dataArray{1} = strtrim(dataArray{1});
dataArray{2} = strtrim(dataArray{2});

%% Close the text file.
fclose(fileID);

%% Convert the contents of columns containing numeric strings to
% numbers. Replace non-numeric strings with NaN.
raw = repmat({''},length(dataArray{1}),length(dataArray)-1);
for col=1:length(dataArray)-1
    raw(1:length(dataArray{col}),col) = dataArray{col};
```

```
end
numericData = NaN(size(dataArray{1},1),size(dataArray,2));

% Converts strings in the input cell array to numbers. Replaced non-
% numeric strings with NaN.
rawData = dataArray{3};
for row=1:size(rawData, 1);
    % Create a regular expression to detect and remove non-numeric
    % prefixes and suffixes.
    regexstr = '(?<prefix>.*?)(?<numbers>([-
]*(\d+[\,]*)+[\.]{0,1}\d*[eEdD]{0,1}[-+]*\d*[i]{0,1})|([-
]*(\d+[\,]*)*[\.]{1,1}\d+[eEdD]{0,1}[-+]*\d*[i]{0,1}))(?<suffix>.*)';
    try
        result = regexp(rawData{row}, regexstr, 'names');
        numbers = result.numbers;

        % Detected commas in non-thousand locations.
        invalidThousandsSeparator = false;
        if any(numbers==',');
            thousandsRegExp = '^\d+?(\,\d{3})*\.{0,1}\d*$';
            if isempty(regexp(numbers, thousandsRegExp, 'once'));
                numbers = NaN;
                invalidThousandsSeparator = true;
            end
        end
        % Convert numeric strings to numbers.
        if ~invalidThousandsSeparator;
            numbers = textscan(strrep(numbers, ',', ''), '%f');
            numericData(row, 3) = numbers{1};
            raw{row, 3} = numbers{1};
        end
    catch me
    end
end


%% Split data into numeric and cell columns.
rawNumericColumns = raw(:, 3);
rawCellColumns = raw(:, [1,2]);


%% Exclude rows with non-numeric cells
I = ~all(cellfun(@(x) (isnumeric(x) || islogical(x)) &&
~isnan(x),rawNumericColumns),2); % Find rows with non-numeric cells
rawNumericColumns(I,:) = [];
rawCellColumns(I,:) = [];

%% Allocate imported array to column variable names
ELEM1 = rawCellColumns(:, 1);
ELEM2 = rawCellColumns(:, 2);
VELLIQ = cell2mat(rawNumericColumns(:, 1));
```

importfaces.m for P2D

```
%
##########################################################################
%NAME    :  importfaces.m
%PURPOSE :  for P2D, import data from file specified in input
%AUTHOR  :  Matteo Scapolo
%
%INPUT   :  faces.txt, startRow = 1, endRow = inf, n
%OUTPUT  :  faces, n_faces
%
##########################################################################
%
%
function [faces, n_faces] = importfile(filename, startRow, endRow, n)
%% Initialize variables.
delimiter = ' ';
if nargin<=2
    startRow = 1;
    endRow = inf;
end

%% Format string for each line of text:
%   column1: double (%f)
% ...
%   column5: double (%f)
%   column6: text (%s)
% ...
%   column42: text (%s)
% For more information, see the TEXTSCAN documentation.
formatSpec =
'%f%f%f%f%f%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%
s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%
s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%
s%[^\n\r]';

%% Open the text file.
fileID = fopen(filename,'r');

%% Read columns of data according to format string.
% This call is based on the structure of the file used to generate
% this code. If an error occurs for a different file, try regenerating
% the code from the Import Tool.
dataArray = textscan(fileID, formatSpec, endRow(1)-startRow(1)+1,
'Delimiter', delimiter, 'MultipleDelimsAsOne', true, 'HeaderLines',
startRow(1)-1, 'ReturnOnError', false);
for block=2:length(startRow)
    frewind(fileID);
    dataArrayBlock = textscan(fileID, formatSpec, endRow(block)-
startRow(block)+1, 'Delimiter', delimiter, 'MultipleDelimsAsOne',
true, 'HeaderLines', startRow(block)-1, 'ReturnOnError', false);
    for col=1:length(dataArray)
        dataArray{col} = [dataArray{col};dataArrayBlock{col}];
    end
end

%% Close the text file.
```

```
fclose(fileID);

%% Post processing for unimportable data.
% No unimportable data rules were applied during the import, so no
% post processing code is included. To generate code which works for
% unimportable data, select unimportable cells in a file and
% regenerate the script.

%% Create output variable
dataArray([1, 2, 3, 4, 5]) = cellfun(@(x) num2cell(x), dataArray([1,
2, 3, 4, 5]), 'UniformOutput', false);
faces = [dataArray{1:end-1}];

i=1; CStr = textread('faces.txt', '%s', 'delimiter', '\n');
for k=1:n
    String=CStr{k};
    Strings=regexp(String, '(?<=\) )\d+(?= \()', 'match');
    n_faces(i,1)=num2cell(str2double(cell2mat(Strings)));
    i=i+1;
end
```

ffaces.m per P2D

```
%
########################################################################
%NAME    :  ffaces.m
%PURPOSE :  for P2D, creation of the matrix VF, containing the
%           components of unit vectors normal to the faces on domain
%           boundaries
%AUTHOR  :  Matteo Scapolo
%
%INPUT   :  k,n_faces, faces, Id
%OUTPUT  :  VF
%
########################################################################
%
%
function [VF] = ffaces(k, n_faces, faces, Id)
% Import of the number of faces from tough2viewer.dat
% Individuation of the column containing the number of faces
    for k5 = 1:size(faces,2)
        tf = strncmpi(num2str(n_faces{k}),faces{k,k5},3);
        if tf == 1
            column_faces = k5;
        end
    end
    % Individuation of all unit vectors from tough2viewer.dat
    for k6 = 1:cell2mat(n_faces(k,1))
        string_vector(k6,:) = faces(k,column_faces +
cell2mat(n_faces(k,1)) + k6);
        VF(k6,:) = sscanf(string_vector{k6},'(%f,%f,%f)',[1 Inf]);
    end
    check_matrix = VF;
    % Elimination from VF of unit vectors with z component different
    % from 0
    VF((VF(:,3)~=0),:) = [];
    % Elimination from VF of unit vectors shared by Id and
    % tough2viewer.dat. Unit vectors remaining are perpendicolar to
    % faces on the domain boundaries
    for cc = 1:size(Id,1)
        if Id(cc,14) == 1
            VF(abs(VF(:,1) - Id(cc,12)) < 0.1...
             & abs(VF(:,2) - Id(cc,13)) < 0.1,:) = [];
        else
            VF(abs(VF(:,1) + Id(cc,12)) < 0.1...
             & abs(VF(:,2) + Id(cc,13)) < 0.1,:) = [];
        end
    end
end
```

Bibliografia

1. Al-Najem A. A., Siddiqui S. e Soliman M. (2012) *Use of streamline simulation in reservoir management*, SPE Saudi Arabia Section Technical Symposium and Exhibition (Al-Khobar, Arabia Saudita, 8-11 aprile 2012).

2. Aurenhammer F. (1991) *Voronoi Diagrams – A survey of a fundamental geometric data structure*, ACM Computing Surveys 23(3), pp: 345 – 405.

3. Attaway S. (2013) *MATLAB A Practical Introduction to Programming and Problem Solving*, Terza Edizione, Oxford, Ed. Elsevier.

4. Brychtova A. e Vozenilek V. (2011) *3Discworld: Automatic Preparation and Visualization of 3D Spatial Data with use of Planar Data*, 25th International Cartography Conference ICC (Parigi, Francia, 3–8 luglio 2011).

5. Bonduà S. et al. (2015) *3D Voronoi pre-and post-processing tools for the modeling of deep sedimentary formations with the TOUGH2 family of codes*, TOUGH Symposium 2015 (Berkeley, California, U.S.A., 28-30 settembre 2015).

6. Bonduà S. et al. (2017) *3D Voronoi grid dedicated software for modeling gas migration in deep layered sedimentary formations with TOUGH2-TMGAS*, Computers & Geosciences, https://doi.org/10.1016/j.cageo.2017.03.008.

7. Bortolotti V. (2014) *Simulazione numerica di bacini serbatoio*, Dispense del corso Modelli Numerici per la Geoingegneria, Alma Mater Studiorum Università di Bologna.

8. Brighenti R. (2014) *Analisi Numerica dei Solidi e delle Strutture: Fondamenti del Metodo degli Elementi finiti*, Bologna, Ed. Esculapio.

9. Ciaburro G. (2000) *Manuale Matlab*, disponibile a: http://www.diee.unica.it/giua/EAS/manuale_MATLAB.pdf, pdf, ultimo accesso 16/06/2017.

10. Citrini D. e Noseda G. (1987) *Idraulica*, Milano, Ed. CEA.

11. Coats K. H. (1987) *Chapter 48 Reservoir Simulation*, in Petroleum Engineering Handbook, H. B. Bradley (a cura di), Richardson, Texas, U.S.A, Ed. Society of Petroleum Engineers.

12. Crane M. J. e Blunt M. J. (1999) *Streamline-based simulation of solute transport*, Water Resources Research 35(10), pp: 2891–3209, doi: 10.1029/1999WR900145.

13. Chierici G. L. (1991) *Principi di ingegneria dei giacimenti petroliferi*, San Donato Milanese, Ed. Agip.

14. De Lucia M. (2008) *Influenza della variabilità spaziale sul trasporto reattivo*, Tesi di Dottorato di Ricerca, Alma Mater Studiorum Università di Bologna, doi: 10.6092/unibo/amsdottorato/856.

15. Finkel M., Liedl R. e Teutsch G. (1999) *Modelling surfactant-enhanced remediation of polycyclic aromatic hydrocarbons*, Environmental Modelling & Software with Environment Data News 14(2-3), pp: 203-211, https://doi.org/10.1016/S1364-8152(98)00071-1.

16. Forgione N. (2001) Soluzione numerica di problemi di flusso di fluidi con contemporaneo scambio di calore e/o di massa, disponibile a: http://www.dimnp.unipi.it/forgione-n/cfd.pdf, pdf, ultimo accesso 29/06/2017.

17. King M.J. e Datta-Gupta A. (1998*) Streamline simulation: a current perspective*, In Situ 22(1), pp: 91–140.

18. Lake L.W. (2007) *Chapter 17 Reservoir Simulation*, Vol. 5 in Petroleum Engineering Handbook, R.P. Batycky et al. (a cura di), Richardson, Texas, U.S.A, Ed. Society of Petroleum Engineers.

19. Marchi E. e Rubatta A. (2004) *Meccanica dei fluidi, principi e applicazioni idrauliche*, Ristampa, Torino, Ed. UTET.

20. Matringe S. F. e Gerritsen M. G. (2004) *On Accurate Tracing of Streamlines*, SPE Annual Technical Conference and Exhibition (Huston, Texas, U.S.A., 26-29 settembre 2004).

21. Matringe S.F. (2008) *Mixed finite element methods for discretization and streamline tracing*, PhD Dissertation, Stanford University.

22. Morel-Seytoux H.J. (1966) Unit mobility ratio displacement calculations for patter floods in homogeneous medium, SPE Journal 13(4), pp: 217-227.

23. Painter S.L., Gable C.W. e Kelkar S. (2012) *Pathline tracing on fully unstructured control-volume grids*, Computational Geosciences 16(4), pp: 1125–1134, doi:10.1007/s10596-012-9307-1.

24. Pollock D.W. (1988) *Semi-analytical computation of path lines for finite-difference models*, Ground Water 26(6), pp: 743–750.

25. Pruess K., Oldenburg C. e Miridis G. (1999) *TOUGH2 User's Guide, Version 2.0*, Earth Sciences Division, Lawrence Berkeley National Laboratory, University of California, Berkeley, California, U.S.A.

26. Thiele M. R. (2001) *Streamline simulation*, 6th International Forum on Reservoir Simulation (Schloss Fuschl, Austria, 3-7 settembre 2001).

27. Tonti E. (2003) *Introduzione a MATLAB*, disponibile a: http://www.discretephysics.org/MANUALI/Matlab.pdf, pdf, ultimo accesso 16/06/2017.

28. Warren J. et al. (2007) *Barycentric coordinates for convex sets*, Advances in Computational Mathematics (27), pp: 319-338, doi: 10.1007/s10444-005-9008-6.

29. Zhang K., Wu Y.S. e Pruess K. (2008) *User's Guide for TOUGH2-MP – A Massively Parallel Version of the TOUGH2 Code*, Earth Sciences Division, Lawrence Berkeley National Laboratory, University of California, Berkeley, California, U.S.A.

Sitografia

30. http://esd1.lbl.gov/research/projects/tough/, ultimo accesso 04/07/2017.

31. https://it.mathworks.com/help/matlab/math/delaunay-triangulation.html, ultimo accesso 28/06/2017.

32. https://it.mathworks.com/products/matlab.html, ultimo accesso 04/07/2017

33. http://math.lbl.gov/voro++/, ultimo accesso 10/07/2017.

34. http://www.afs.enea.it/funel/CFD/OpenFOAMDocumentation/meshconvita.pdf, ultimo accesso 28/06/2017.

35. https://www.gnu.org/software/octave/, ultimo accesso 04/07/2017.

36. http://www.treccani.it/enciclopedia/simulazione, ultimo accesso 29/06/2017.

37. http://www.treccani.it/enciclopedia/sistema/, ultimo accesso 28/06/2017.