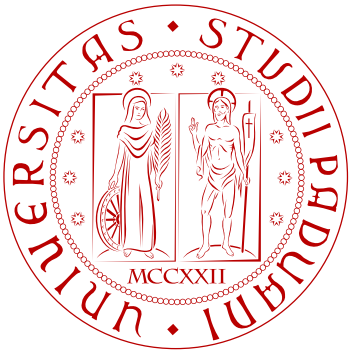
UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA CIVILE, EDILE E AMBIENTALE
Department of Civil, Environmental and Architectural Engineering

Corso di Laurea Magistrale in Ingegneria Civile



TESI DI LAUREA

GReS: General Reservoir Simulator - Sviluppo di un simulatore geomeccanico di giacimenti in Matlab

Relatore: Ch.mo Prof. Massimiliano Ferronato
Correlatore: Dott. Andrea Franceschini

Laureando: Angelica Righetto
Matricola: 1220412

ANNO ACCADEMICO 2021–2022

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

Il presente lavoro si incentra sullo sviluppo preliminare di uno strumento generico per la simulazione di problemi di giacimento, che riguardano lo stoccaggio di fluidi nel sottosuolo. Con stoccaggio di fluidi si intende, ad esempio, l'iniezione di CO_2 per ridurne l'emissione nell'atmosfera (figura 1.1) o l'immagazzinamento di idrogeno, in alternativa all'impiego di serbatoi. L'iniezione e l'estrazione di fluidi nel sottosuolo comportano una variazione dello stato tensionale nell'intorno del giacimento e, di conseguenza, l'innesco di un processo di assestamento del terreno. La previsione dell'entità di deformazioni e spostamenti causate dai suddetti processi risulta necessaria a prevenire fenomeni che potrebbero avere conseguenze di natura ambientale e socioeconomica, come ad esempio la subsidenza o la riattivazione delle faglie.

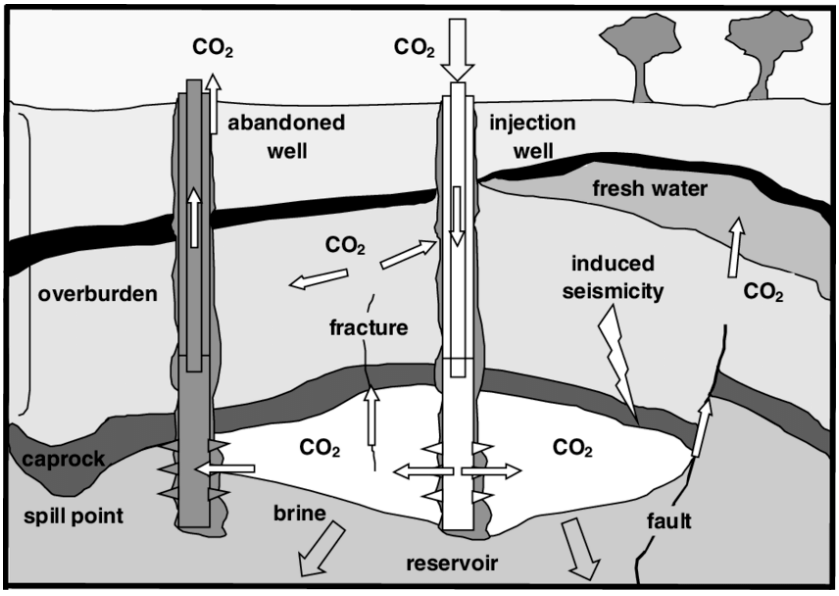


Figura 1.1: Stoccaggio di CO_2 in un giacimento.

GReS (General Reservoir Simulator) è un simulatore generale di problemi di giacimento in grado di considerare molteplici fenomeni fisici che possono essere coinvolti durante i processi di stoccaggio di fluidi nel sottosuolo. L'obbiettivo del codice, a differenza degli altri

software commerciali, è quindi quello di affrontare generalmente un giacimento, considerando anche fisiche che risultano ancora oggi oggetto di ricerca e sviluppo.
Il presente lavoro di tesi si è concentrato sulla risoluzione del problema dell'equilibrio elastico per valutare l'entità degli spostamenti che si sviluppano nell'area interessata dal giacimento. Per fare questo, si è definito il modello matematico che descrive la fisica del fenomeno e si è risolto il sistema di equazioni differenziali che ne deriva tramite un modello numerico. In particolare, il modello matematico si basa sul principio dei lavori virtuali per il quale, in presenza di una configurazione deformata congruente con i vincoli, l'uguaglianza tra lavoro virtuale interno e lavoro virtuale esterno è condizione necessaria e sufficiente per l'equilibrio del sistema. Per quanto riguarda il modello numerico, lo studio avviene per mezzo del metodo degli elementi finiti. È prevista, dunque, la discretizzazione del dominio di interesse in elementi finiti tridimensionali, la definizione delle condizioni al contorno necessarie alla risoluzione del problema differenziale e la scelta del modello costitutivo del materiale. Quest'ultimo condiziona il legame tra le variazioni di tensione e le variazioni di deformazione e, di conseguenza, l'entità degli spostamenti nel giacimento.

GReS è sviluppato in Matlab, utilizzando la programmazione ad oggetti, con lo scopo di realizzare uno strumento flessibile. La programmazione a classi garantisce, infatti, lo sviluppo di uno strumento modulare che facilita il processo di implementazione, anche da parte di più programmatori contemporaneamente, e il processo di integrazione del codice, così da ottenere uno strumento sempre più completo ed efficiente. Si è scelto di realizzare il codice in Matlab perché è un linguaggio di programmazione ad alto livello, molto diffuso e con un basso costo di accesso. Inoltre, può richiamare routine sviluppate in linguaggi di programmazione di basso livello, in modo tale da preservare l'efficienza computazionale del codice.

Il presente lavoro di tesi si incentra, inizialmente, sugli aspetti teorici legati allo sviluppo del codice e termina con lo svolgimento di una serie di test di validazione. Nel secondo capitolo vengono presentate le equazioni differenziali che governano il problema strutturale dell'equilibrio, la formulazione debole che ne deriva e la descrizione del metodo degli elementi finiti per il calcolo della soluzione approssimata del problema. Il terzo capitolo illustra i modelli costitutivi dei materiali implementati finora, ovvero il modello elastico lineare isotropo, il modello elastico lineare trasversalmente isotropo e il modello ipoelastico. Il quarto capitolo tratta la tecnica della programmazione a classi e descrive la struttura delle diverse classi implementate. Nel quinto capitolo si riportano i test di validazione del codice e, infine, l'ultimo capitolo è dedicato alle conclusioni del presente lavoro e agli sviluppi futuri di GReS. La stesura del codice è riportata in appendice.

Capitolo 2

Modello matematico-numerico

L'entità delle deformazioni generate da processi di iniezione e/o estrazione di fluidi nel sottosuolo è calcolata mediante un modello matematico basato sulle equazioni indefinite di equilibrio. Il problema consiste nel trovare la configurazione deformata del dominio $\Omega \in \mathbb{R}^3$ tale da equilibrare le forze esterne applicate e rispettante i vincoli imposti. Prima di presentare le equazioni da risolvere, verranno brevemente introdotti alcuni concetti utili nel seguito, come i tensori di deformazione e tensione.

2.1 Tensore della deformazione

In un sistema di riferimento cartesiano, il tensore della deformazione relativo ad un punto appartenente al continuo tridimensionale è definito come:

$$\varepsilon = \frac{1}{2}\left(\nabla s + \nabla^T s\right) = \begin{bmatrix} \varepsilon_{xx} & \gamma_{xy} & \gamma_{xz} \\ \gamma_{yx} & \varepsilon_{yy} & \gamma_{yz} \\ \gamma_{zx} & \gamma_{zy} & \varepsilon_{zz} \end{bmatrix} \tag{2.1}$$

Le componenti del tensore poste sulla diagonale rappresentano le variazioni di lunghezza delle fibre disposte secondo le tre direzioni del sistema di riferimento x, y e z. Le restanti componenti rappresentano, invece, gli scorrimenti angolari delle fibre ortogonali e disposte secondo le direzioni x,y e y,z e z,x. Le componenti di deformazione sono definite come le derivate parziali delle componenti di spostamento che il corpo rigido matura durante il processo deformativo, ovvero il processo di passaggio da una configurazione iniziale indeformata ad una configurazione deformata. Il vettore spostamento è definito come:

$$\mathbf{u} = [u\ v\ w]^T \tag{2.2}$$

e le sue componenti sono la misura della distanza tra un punto facente parte della configurazione indeformata e lo stesso punto facente parte della configurazione deformata. In accordo con l'ipotesi di piccoli spostamenti e piccole deformazioni, le componenti complementari

relative allo scorrimento angolare vengono sommate ottenendo:

$$\begin{aligned}
\varepsilon_x &= \frac{\partial u}{\partial x} \\
\varepsilon_y &= \frac{\partial v}{\partial y} \\
\varepsilon_z &= \frac{\partial w}{\partial z} \\
\gamma_{xy} = \gamma_{yx} &= \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \\
\gamma_{yz} = \gamma_{zy} &= \frac{\partial v}{\partial z} + \frac{\partial w}{\partial y} \\
\gamma_{zx} = \gamma_{xz} &= \frac{\partial w}{\partial x} + \frac{\partial u}{\partial z}
\end{aligned} \tag{2.3}$$

Si noti che il tensore della deformazione è simmetrico per costruzione.

Utilizzando la notazione di Voigt, il tensore simmetrico è equivalente al vettore:

$$\varepsilon = [\varepsilon_x \ \varepsilon_y \ \varepsilon_z \ \gamma_{xy} \ \gamma_{yz} \ \gamma_{zx}]^T \tag{2.4}$$

2.2 Tensore della tensione

In un sistema di riferimento cartesiano, il tensore della tensione relativo ad un punto appartenente al continuo tridimensionale è definito come:

$$\sigma = \begin{bmatrix} \sigma_{xx} & \tau_{xy} & \tau_{xz} \\ \tau_{yx} & \sigma_{yy} & \tau_{yz} \\ \tau_{zx} & \tau_{zy} & \sigma_{zz} \end{bmatrix} \tag{2.5}$$

Le componenti del tensore poste sulla diagonale sono dette *componenti normali* di tensione, mentre le restanti componenti sono dette *componenti tangenziali*. Secondo il teorema di reciprocità tra tensioni tangenziali, le componenti tangenziali del tensore σ sono simmetriche:

$$\begin{aligned}
\tau_{xy} &= \tau_{yx} \\
\tau_{xz} &= \tau_{zx} \\
\tau_{yz} &= \tau_{zy}
\end{aligned} \tag{2.6}$$

A questo punto, il tensore è esprimibile anche in forma vettoriale come:

$$\sigma = [\sigma_x \ \sigma_y \ \sigma_z \ \tau_{xy} \ \tau_{yz} \ \tau_{zx}]^T \tag{2.7}$$

Le componenti di tensione con segno positivo fanno riferimento ad una trazione, mentre le componenti con segno negativo fanno riferimento ad una compressione del mezzo.

2.3 Problema dell'equilibrio elastico

Si consideri un continuo tridimensionale in regime di spostamenti infinitesimi, caratterizzato da un modello costitutivo che lega le componenti del tensore delle tensioni alle componenti del tensore di deformazione, ed è soggetto a forze di volume $\mathbf{f} = [f_x,\ f_y,\ f_z]^T$ agenti in Ω e di superficie $\mathbf{t} = [t_x,\ t_y,\ t_z]^T$ agenti in $\partial\Omega_t$. Il corpo è inoltre soggetto a dei vincoli in $\partial\Omega_u$. La risoluzione del *problema dell'equilibrio elastico* consiste nella determinazione delle sei componenti del tensore delle tensioni (2.7), delle sei componenti del tensore delle deformazioni (2.4) e delle tre componenti del vettore spostamento (2.2). Le equazioni che governano il problema elastico sono le equazioni di congruenza (2.3), le equazioni costitutive, che dipendono dal tipo di legame assunto per il materiale, e le equazioni indefinite di equilibrio di Cauchy:

$$\begin{aligned} &\frac{\partial \sigma_x}{\partial x} + \frac{\partial \tau_{xy}}{\partial y} + \frac{\partial \tau_{xz}}{\partial z} + f_x = 0 \\ &\frac{\partial \tau_{yx}}{\partial x} + \frac{\partial \sigma_y}{\partial y} + \frac{\partial \tau_{yz}}{\partial z} + f_y = 0 \qquad in\ \Omega \\ &\frac{\partial \tau_{zx}}{\partial x} + \frac{\partial \tau_{zy}}{\partial y} + \frac{\partial \sigma_z}{\partial z} + f_z = 0 \end{aligned} \tag{2.8}$$

con le relative condizioni al contorno sulla superficie non vincolata:

$$\begin{aligned} &\alpha\sigma_x + \beta\tau_{yx} + \gamma\tau_{zx} = t_x \\ &\alpha\tau_{xy} + \beta\sigma_y + \gamma\tau_{zy} = t_y \qquad in\ \partial\Omega_t \\ &\alpha\tau_{xz} + \beta\tau_{yz} + \gamma\sigma_z = t_z \end{aligned} \tag{2.9}$$

in cui α, β e γ sono i coseni direttori della direzione normale alla superficie.

Dato che la soluzione analitica di suddette equazioni non è nota, si sceglie di risolvere il sistema di equazioni differenziali tramite il *metodo degli elementi finiti*, un metodo variazionale che approssima la soluzione esatta del problema differenziale. I metodi variazionali sono ottimali quando esiste un principio variazionale associato, ovvero quando la ricerca della soluzione è riconducibile alla minimizzazione di una forma integrale, detta *funzionale*, associata al principio stesso. Nel caso del problema dell'equilibrio elastico, la soluzione si ottiene minimizzando il funzionale associato al principio variazionale che esprime l'energia potenziale totale:

$$\Psi(\mathbf{u}) = U + W \tag{2.10}$$

in cui U rappresenta l'energia elastica e W l'energia potenziale delle forze agenti sul volume e sulla superficie del corpo. La derivata prima dell'energia potenziale diventa un contributo nel principio dei lavori virtuali. Secondo tale principio, in presenza di una configurazione deformata congruente con i vincoli, l'uguaglianza tra lavoro virtuale interno e lavoro virtuale esterno è condizione necessaria e sufficiente per l'equilibrio del sistema. Il principio dei lavori virtuali deriva direttamente dalla minimizzazione dell'energia potenziale del sistema

e si può scrivere matematicamente come:

$$\int_{\Omega}(f_x u^v + f_y v^v + f_z w^v)\,d\Omega + \int_{\partial\Omega_t}(t_x u^v + t_y v^v + t_z w^v)\,d\Omega_t = \int_{\Omega}\sum_{i,j}\sigma_{ij}\varepsilon_{ij}^v\,d\Omega \tag{2.11}$$

in cui il pedice v indica le componenti virtuali. La formulazione integrale del principio dei lavori virtuali è la formulazione debole del sistema differenziale che si utilizzerà per applicare il metodo di Galerkin. Il metodo di Galerkin è un metodo variazionale ovvero un metodo in cui si cerca la soluzione approssimata di un problema differenziale all'interno di uno sottospazio funzionale che è contenuto nello spazio funzionale a cui appartiene la soluzione esatta. In particolare, il metodo di Galerkin si basa sulla minimizzazione del residuo.

2.4 Metodo degli elementi finiti

Nel risolvere il sistema di equazioni differenziali formulato a partire dal problema dell'equilibrio elastico, si è impiegato il metodo degli elementi finiti, in particolare il metodo di Galerkin. Nel metodo degli elementi finiti le funzioni di base degli elementi N_i sono dei *polinomi di interpolazione continui a tratti a supporto locale* [Gambolati and Ferronato, 2015]. La soluzione approssimata $\hat{u}_n$ del problema è una combinazione lineare delle funzioni di base:

$$\hat{u}_n = \alpha_1 N_1(\mathbf{x}) + \alpha_2 N_2(\mathbf{x}) + \ldots + \alpha_n N_n(\mathbf{x}) = \sum_{i=1}^{n}\alpha_i N_i(\mathbf{x}) \tag{2.12}$$

In generale, i coefficienti α_i devono minimizzare la norma energia dell'errore e_n:

$$e_n = \hat{u}_n - \bar{u} \tag{2.13}$$

in cui $\bar{u}$ rappresenta la soluzione esatta del problema differenziale. Il metodo degli elementi finiti è un metodo in cui coefficienti α_i sono direttamente i valori della funzione incognita sugli n nodi del dominio discretizzato Ω. Una volta scelte le funzioni di base, l'accuratezza della soluzione dipende dal numero di nodi del dominio discretizzato.

Volendo ricavare le equazioni degli elementi finiti nel caso dell'equilibrio elastico di una struttura tridimensionale, si inizia col definire le componenti del campo di spostamento, riferite all'elemento e della mesh. In generale:

$$\begin{aligned}
u^{(e)}(x,y,z) &= u_1 N_1^{(e)}(x,y,z) + u_2 N_2^{(e)}(x,y,z) + \ldots + u_n N_n^{(e)}(x,y,z)\\
v^{(e)}(x,y,z) &= v_1 N_1^{(e)}(x,y,z) + v_2 N_2^{(e)}(x,y,z) + \ldots + v_n N_n^{(e)}(x,y,z)\\
w^{(e)}(x,y,z) &= w_1 N_1^{(e)}(x,y,z) + w_2 N_2^{(e)}(x,y,z) + \ldots + w_n N_n^{(e)}(x,y,z)
\end{aligned} \tag{2.14}$$

in cui n è l'indice di nodo dell'elemento. In ipotesi di deformazioni infinitesime, il tensore

delle deformazioni è composto da sei componenti e si può scrivere come:

$$\boldsymbol{\varepsilon}^{(e)} = \begin{bmatrix} \varepsilon_x^{(e)} \\ \varepsilon_y^{(e)} \\ \varepsilon_z^{(e)} \\ \gamma_{xy}^{(e)} \\ \gamma_{yz}^{(e)} \\ \gamma_{zx}^{(e)} \end{bmatrix} = \begin{bmatrix} \frac{\partial u^{(e)}}{\partial x} \\ \frac{\partial v^{(e)}}{\partial y} \\ \frac{\partial w^{(e)}}{\partial z} \\ \frac{\partial u^{(e)}}{\partial y} + \frac{\partial v^{(e)}}{\partial x} \\ \frac{\partial v^{(e)}}{\partial z} + \frac{\partial w^{(e)}}{\partial y} \\ \frac{\partial w^{(e)}}{\partial x} + \frac{\partial u^{(e)}}{\partial z} \end{bmatrix} \tag{2.15}$$

Posto il vettore degli spostamenti $\mathbf{u}^{(e)} = [u^{(e)},\ v^{(e)},\ w^{(e)}]^T$ e il vettore delle componenti di spostamento sui nodi $\mathbf{s}^{(e)} = [u_i,\ v_i,\ w_i,\ ...,\ u_n,\ v_n,\ w_n]^T$, si ha:

$$\mathbf{u}^{(e)} = \begin{bmatrix} N_i^{(e)} & 0 & 0 & N_j^{(e)} & 0 & 0 & \dots & N_n^{(e)} & 0 & 0 \\ 0 & N_i^{(e)} & 0 & 0 & N_j^{(e)} & 0 & \dots & 0 & N_n^{(e)} & 0 \\ 0 & 0 & N_i^{(e)} & 0 & 0 & N_j^{(e)} & \dots & 0 & 0 & N_n^{(e)} \end{bmatrix} \mathbf{s}^{(e)} = N^{(e)}\mathbf{s}^{(e)} \tag{2.16}$$

Il tensore delle deformazioni si scrive come:

$$\boldsymbol{\varepsilon}^{(e)} = B^{(e)}\mathbf{s}^{(e)} \tag{2.17}$$

in cui $B^{(e)}$ è la matrice locale, cioè riferita all'elemento, che lega il tensore di deformazione alle componenti di spostamento sui nodi e raccoglie le derivate delle funzioni di base. Assumendo che il corpo sia indeformato nella condizione iniziale e indicando con D la matrice costitutiva del materiale, la relazione che lega il tensore della tensione al tensore della deformazione è il seguente:

$$\boldsymbol{\sigma}^{(e)} = D^{(e)}(\boldsymbol{\sigma})\boldsymbol{\varepsilon} \tag{2.18}$$

Si indicano, ora, il vettore degli spostamenti nodali virtuali con $\mathbf{s}^{(e),v}$ e il vettore degli spostamenti virtuali con $\mathbf{u}^{(e),v}$. Il lavoro virtuale delle componenti ordinate delle eventuali forze concentrate sui nodi dell'elemento è:

$$L_1^{(e),v} = \left(\mathbf{s}^{(e),v}\right)^T F^{(e)} \tag{2.19}$$

Se $\mathbf{f}^{(e)} = [f_x^{(e)}, f_y^{(e)}, f_z^{(e)}]^T$ è il vettore delle componenti delle forze di massa, il lavoro di queste ultime è:

$$L_2^{(e),v} = \int_{\Omega_e} \left(\mathbf{u}^{(e),v}\right)^T \mathbf{f}^{(e)}\, d\Omega = \left(\mathbf{s}^{(e),v}\right)^T \int_{\Omega_e} N^{(e),T}\mathbf{f}^{(e)}\, d\Omega \tag{2.20}$$

Il lavoro delle forze virtuali interne, invece, si scrive:

$$L_3^{(e),v} = \int_{\Omega_e} \left(\boldsymbol{\varepsilon}^{(e),v}\right)^T \boldsymbol{\sigma}^{(e)}\, d\Omega \tag{2.21}$$

Se si sostituiscono la (2.17) e la (2.18) all'interno della definizione precedente, si ottiene:

$$L_3^{(e),v} = \int_{\Omega_e} \left(\mathbf{s}^{(e),v}\right)^T B^{(e),T} D^{(e)} B^{(e)} \mathbf{s}^{(e)} \, d\Omega = \left(\mathbf{s}^{(e),v}\right)^T \left[\int_{\Omega_e} B^{(e),T} D^{(e)} B^{(e)} \, d\Omega\right] \mathbf{s}^{(e)} \tag{2.22}$$

Per il principio dei lavori virtuali, espresso in (2.11), uguagliando il lavoro delle forze interne alla somma del lavoro delle forze esterne e delle forze di massa, si ottiene:

$$\left(\mathbf{s}^{(e),v}\right)^T \left[\int_{\Omega_e} B^{(e),T} D^{(e)} B^{(e)} \, d\Omega\right] \mathbf{s}^{(e)} = \left(\mathbf{s}^{(e),v}\right)^T F^{(e)} + \left(\mathbf{s}^{(e),v}\right)^T \int_{\Omega_e} N^{(e),T} \mathbf{f}^{(e)} \, d\Omega \tag{2.23}$$

Dato che l'equazione deve valere per ogni spostamento virtuale, si ottiene:

$$\left[\int_{\Omega_e} B^{(e),T} D^{(e)} B^{(e)} \, d\Omega\right] \mathbf{s}^{(e)} = F^{(e)} + \int_{\Omega_e} N^{(e),T} \mathbf{f}^{(e)} \, d\Omega \tag{2.24}$$

L'equazione (2.24) rappresenta il contributo dell'elemento e al principio dei lavori virtuali del sistema. In generale, la matrice di rigidezza di un elemento finito tridimensionale è data da:

$$H^{(e)} = \int_{\Omega_e} B^{(e),T} D^{(e)} B^{(e)} \, d\Omega \tag{2.25}$$

La matrice di rigidezza dell'elemento è una matrice quadrata la cui dimensione è data dal prodotto tra il numero di nodi e il numero di gradi di libertà assegnati a ciascun nodo. Assemblando i contributi di rigidezza valutati in ciascun elemento della mesh tridimensionale, si ottiene il sistema:

$$H(\mathbf{s})\mathbf{s} = \mathbf{r} \tag{2.26}$$

in cui:

- H è la matrice di rigidezza globale del sistema;
- **s** è il vettore degli spostamenti nodali incogniti;
- **r** è il termine noto che comprende l'effetto delle forze di massa e l'effetto delle forze applicate ai nodi.

Per la risoluzione del sistema (2.26) è necessario eliminare i moti rigidi della struttura mediante appropriate condizioni al contorno di Dirichlet. Dopo aver calcolato gli spostamenti nodali mediante la soluzione del sistema (2.26), è possibile valutare anche le tensioni all'interno della struttura:

$$\boldsymbol{\sigma}^{(e)} = D^{(e)} B^{(e)} \mathbf{s}^{(e)} \tag{2.27}$$

Si noti che se la matrice $D^{(e)}$ è simmetrica e definita positiva, di conseguenza anche $H^{(e)}$ è simmetrica e definita positiva e la soluzione ottenuta con il metodo variazionale è ottimale.

La risoluzione del sistema con il metodo degli elementi finiti prevede le seguenti fasi:

1. *Discretizzazione del dominio*, ovvero il passaggio da un sistema con infiniti gradi di libertà, ad un sistema discreto caratterizzato da un numero finito di incognite;

2. *Scelta delle funzioni di base*, in particolare, del grado di queste ultime;

3. *Definizione del modello costitutivo del materiale*, ovvero del legame tra le componenti di tensione e le componenti di deformazione;

4. *Assemblaggio* della matrice di rigidezza globale H;

5. *Imposizione delle condizioni al contorno*, necessarie al calcolo della soluzione approssimata del problema fisico, congruente con i vincoli e con le forze applicate;

6. *Soluzione del sistema*:

$$H(\mathbf{s})\mathbf{s} = \mathbf{r}$$

2.4.1 Definizione della griglia

Il processo di definizione della griglia comprende una prima fase di discretizzazione del dominio e una seconda fase di imposizione delle condizioni al contorno, variabili a seconda del problema analizzato. Il processo di discretizzazione consiste nella scelta del tipo di elemento finito che si vuole impiegare, nella scelta del grado delle funzioni di base dell'elemento e nella suddivisione del continuo, senza che vi siano sovrapposizioni tra gli elementi. Il processo di discretizzazione deve garantire un adeguato grado di accuratezza della soluzione.

Esistono diversi tipi di elementi finiti (elementi asta, elementi bidimensionali, elementi tridimensionali) e la scelta dell'elemento si basa sulla geometria del dominio da discretizzare. GReS è un simulatore di problemi di geo-meccanica, di conseguenza il dominio viene suddiviso in elementi tridimensionali, come ad esempio gli esaedri (indicati per i modelli caratterizzati da una geometria regolare) e i tetraedri. Per uno stesso elemento esistono poi diversi gradi di approssimazione della soluzione. Infatti, un elemento tetraedrico può essere di tipo lineare (4 nodi) o quadratico (10 nodi), a seconda del grado delle sue funzioni di base.

Gli elementi della mesh, ovvero del dominio discretizzato, sono sempre univocamente individuati da una sequenza di nodi, denominata *topologia*. Inoltre, ad ogni elemento vengono attribuiti uno o più indici, che possono ricondurre al materiale assegnato all'elemento, o alla regione del dominio in cui è posizionato. Una mesh può quindi essere composta da diversi materiali, i cui parametri cambiano di zona in zona. I nodi, invece, sono delle entità puntuali, la cui posizione nello spazio è univocamente definita da un set di coordinate.

Una volta generato il modello discreto, è necessario imporre opportune condizioni ausiliarie, altrimenti, l'equazione differenziale che descrive il problema fisico possiede un'infinità di soluzioni.

Le condizioni ausiliarie possono riguardare i valori della funzione e/o i valori della derivata normale sul contorno. È necessario imporre un set adeguato di condizioni al contorno per avere un *problema ben posto*, ovvero un problema per il quale la soluzione è unica e dipende in maniera continua dai dati e dalle condizioni ausiliarie stesse. Per un problema stazionario, le condizioni ausiliarie relative ad un dominio tridimensionale si possono scrivere

come:

$$\alpha(x,y,z)\mathbf{u}(x,y,z) + \beta(x,y,z)\frac{\partial \mathbf{u}}{\partial \mathbf{n}}(x,y,z) = \gamma(x,y,z) \quad \forall\, x,y,z \in \partial\Omega \tag{2.28}$$

in cui:

- α, β, γ sono funzioni assegnate dello spazio;
- $\frac{\partial \mathbf{u}}{\partial \mathbf{n}}$ è la derivata normale alla frontiera nello spazio.

Quando $\beta = 0$ si parla di condizioni al contorno *principali* o di *Dirichlet*, e corrispondono all'assegnare un valore alla soluzione incognita su tutto o su parte del dominio $\partial\Omega$. Quando $\alpha = 0$ invece, si parla di condizioni al contorno iniziali *naturali* o di *Neumann*, per le quali si definisce il valore della derivata normale della soluzione incognita su parte del dominio $\partial\Omega$. Se $\beta \neq 0$ e anche $\alpha \neq 0$, si parla di condizioni al contorno *miste* o di *Cauchy*. Se $\gamma = 0$ le condizioni ausiliarie si dicono omogenee.

Le condizioni di Dirichlet trasformano la matrice di rigidezza globale del sistema, H, in una matrice non singolare. Per arrivare ad una matrice regolare sarebbe necessario imporre un valore unitario a tutti i termini diagonali corrispondenti ai gradi di libertà vincolati con le condizioni di Dirichlet, azzerare tutti gli elementi extra-diagonali della riga e della colonna corrispondenti ed eguagliare il termine noto, $\mathbf{r}_i$, al valore della condizioni di Dirichlet. Tuttavia, questo procedimento risulta piuttosto complesso, perciò all'interno di GReS si è scelto di imporre le condizioni di Dirichlet tramite il metodo *penalty*. Questo metodo consiste nell'assegnare a tutti i termini diagonali corrispondenti ai gradi di libertà vincolati con le condizioni di Dirichlet un valore molto elevato (circa 10 ordini di grandezza maggiore rispetto al valore del termine massimo della matrice di rigidezza) ed uguagliare il termine noto corrispondente al valore della condizioni di Dirichlet, aumentato anch'esso dello stesso numero di ordini di grandezza. Per quanto riguarda le condizioni di Neumann, l'imposizione di queste condizioni comporta la modifica del termine noto, $\mathbf{r}$, in corrispondenza dei gradi di libertà vincolati.

2.4.2 Elementi finiti

Il metodo degli elementi finiti prevede la suddivisione del dominio Ω in elementi, sui cui nodi viene calcolata la soluzione approssimata. A ciascun elemento viene associata una funzione, detta *funzione base*, ovvero un polinomio a supporto locale, che assume valore unitario solo in corrispondenza del nodo a cui è riferito e risulta nulla nel resto del dominio.

In particolare, gli elementi finiti tridimensionali sono elementi le cui funzioni di base sono il prodotto di polinomi di interpolazione locale, di grado variabile. La scelta del grado della funzione base può essere condizionata dal tipo di problema fisico che si intende risolvere e dal grado di approssimazione locale richiesto per la soluzione. Ad esempio, la risoluzione di un problema potrebbe richiedere il calcolo della derivata seconda, a quel punto sarebbe necessario l'impiego di elementi finiti *quadratici*, piuttosto che *lineari*.

Tuttavia, sebbene il polinomio interpolatore dell'elemento possa aumentare di grado, tutti gli elementi finiti sono elementi di classe C^0, il che significa che le funzioni di base non sono derivabili sui nodi e il grado di approssimazione della soluzione globale non può essere aumentato. La scelta della funzione di base può essere condizionata anche dal costo computazionale associato a ciascun elemento, che cresce al crescere del grado del polinomio interpolatore.

All'interno di GReS sono stati implementati due tipologie di elemento finito tridimensionale, l'elemento tetraedrico lineare e l'elemento esaedrico trilineare.

Tetraedri lineari

I tetraedri lineari sono elementi tridimensionali caratterizzati da *4 nodi*, con funzioni di forma lineari. La loro geometria permette di discretizzare in maniera piuttosto precisa qualsiasi dominio, anche quelli di forma più irregolare. Inoltre, non necessitano dell'integrazione della matrice di rigidezza dato che le derivate delle funzioni di forma sono costanti all'interno del volume dell'elemento. Tuttavia, questo aspetto rende il loro uso limitato, in quanto il calcolo delle deformazioni e delle tensioni può risultare inaccurato.

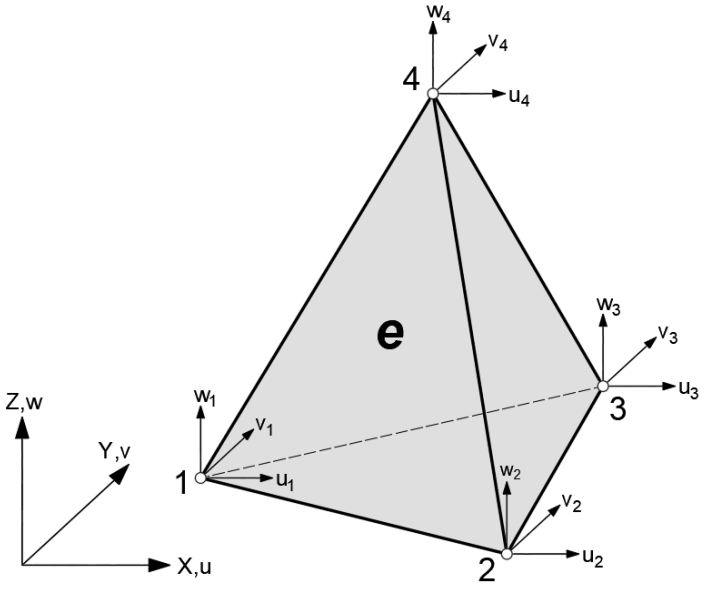


Figura 2.1: Elemento finito tetraedrico lineare a 4 nodi. Tre gradi di libertà per ciascun nodo.

L'elemento presenta sei lati e quattro facce, ciascun lato è individuato da due nodi e ciascuna faccia è individuata da tre nodi (figura 2.1). Ad ogni nodo dell'elemento sono associati 3 gradi di libertà che, nel caso del problema elastico, corrispondono alle componenti di spostamento lungo le tre direzioni del sistema di riferimento.

Le funzioni di base dell'elemento N_i sono lineari e sono definite in generale come:

$$N_i^{(e)}(x,y,z) = \frac{a_i + b_i x + c_i y + d_i z}{6V^{(e)}} \qquad con\ i = 1,2,3,4 \tag{2.29}$$

$V^{(e)}$ è il volume con segno dell'elemento, pari a:

$$V^{(e)} = \frac{1}{6}\begin{vmatrix} 1 & x_1 & y_1 & z_1 \\ 1 & x_2 & y_2 & z_2 \\ 1 & x_3 & y_3 & z_3 \\ 1 & x_4 & y_4 & z_4 \end{vmatrix} \tag{2.30}$$

e a_i, b_i, c_i, d_i sono dei coefficienti. Per il nodo 1 si calcolano come:

$$\begin{aligned} a_1 &= \begin{vmatrix} x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \\ x_4 & y_4 & z_4 \end{vmatrix} & b_1 &= -\begin{vmatrix} 1 & y_2 & z_2 \\ 1 & y_3 & z_3 \\ 1 & y_4 & z_4 \end{vmatrix} \\ c_1 &= \begin{vmatrix} 1 & x_2 & z_2 \\ 1 & x_3 & z_3 \\ 1 & x_4 & z_4 \end{vmatrix} & d_1 &= -\begin{vmatrix} 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \\ 1 & x_4 & y_4 \end{vmatrix} \end{aligned} \tag{2.31}$$

Per il calcolo dei coefficienti sugli altri nodi, basta permutare gli indici. Le funzioni di base sono polinomi a supporto locale, quindi devono soddisfare necessariamente due condizioni:

$$N_i^{(e)}(x,y,z) = \begin{cases} 1 \; se \; i = j \\ 0 \; se \; i \neq j \end{cases} \tag{2.32}$$

$$\sum_{i=1}^{4} N_i^{(e)}(x,y,z) = 1 \tag{2.33}$$

Considerando il tetraedro in figura 2.1, le componenti dello spostamento, ovvero le incognite del problema, si possono scrivere in termini di funzioni di base come:

$$\begin{aligned} u^{(e)}(x,y,z) &= u_1 N_1^{(e)}(x,y,z) + u_2 N_2^{(e)}(x,y,z) + u_3 N_3^{(e)}(x,y,z) + u_4 N_4^{(e)}(x,y,z) \\ v^{(e)}(x,y,z) &= v_1 N_1^{(e)}(x,y,z) + v_2 N_2^{(e)}(x,y,z) + v_3 N_3^{(e)}(x,y,z) + v_4 N_4^{(e)}(x,y,z) \\ w^{(e)}(x,y,z) &= w_1 N_1^{(e)}(x,y,z) + w_2 N_2^{(e)}(x,y,z) + w_3 N_3^{(e)}(x,y,z) + w_4 N_4^{(e)}(x,y,z) \end{aligned} \tag{2.34}$$

La *matrice delle derivate delle funzioni di base B* ha dimensioni [6x12]:

$$B^{(e)} = [B_1{}^{(e)} \; B_2{}^{(e)} \; B_3{}^{(e)} \; B_4{}^{(e)}] \tag{2.35}$$

in cui la matrice delle derivate riferita al nodo i-esimo (con *i = 1, 2, 3, 4*) è pari a:

$$B_i^{(e)}(x,y,z) = \frac{1}{6V^{(e)}} \begin{bmatrix} b_i & 0 & 0 \\ 0 & c_i & 0 \\ 0 & 0 & d_i \\ c_i & b_i & 0 \\ 0 & d_i & c_i \\ d_i & 0 & b_i \end{bmatrix} \tag{2.36}$$

In generale, la matrice di rigidezza di un elemento finito tridimensionale è data dall'equazione (2.25) e necessita del calcolo di un integrale sul volume. Per i tetraedri lineari, tuttavia, la matrice delle derivate delle funzioni di base B e la matrice del legame costitutivo D sono costanti su tutto l'elemento, dunque la matrice di rigidezza può essere calcolata più semplicemente come:

$$H^{(e)} = B^{(e),T} D^{(e)} \; B^{(e)} \; V^{(e)} \tag{2.37}$$

senza la necessità dell'impiego di metodi di integrazione. La matrice di rigidezza locale di un tetraedro lineare ha dimensione [12x12].

Esaedri trilineari

L'elemento esaedrico, chiamato anche elemento *brick*, è un elemento finito tridimensionale di forma prismatica. In particolare l'esaedro trilineare è un elemento definito da *8 nodi* posti ai vertici e sei facce. Le funzioni di forma sono ottenute dal prodotto di tre polinomi lineari di Lagrange. Per questo motivo, questo elemento è detto *trilineare*, ovvero separatamente lineare in ognuna delle tre variabili del sistema di riferimento.

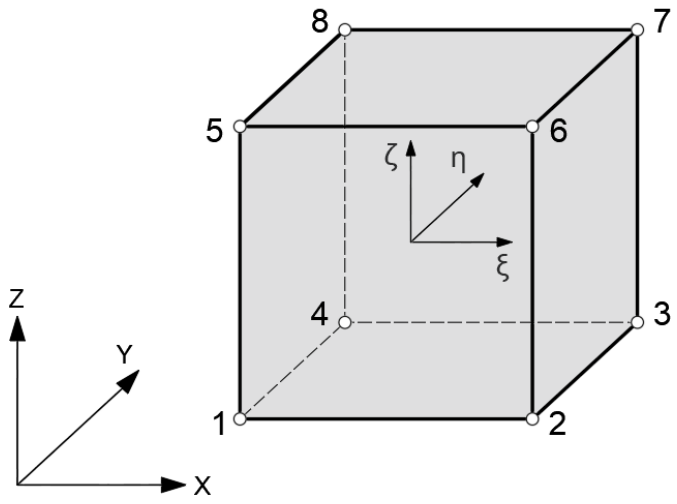


Figura 2.2: Elemento finito esaedrico trilineare a 8 nodi. Tre gradi di libertà per ciascun nodo.

Per facilitare il calcolo degli integrali presenti nella trattazione dell'elemento esaedrico trilineare, si effettua un cambio di variabili mediante il quale l'elemento viene trasformato in un cubo di geometria *di riferimento* (figura 2.2). Questo cubo è centrato nell'origine di un sistema di riferimento locale di coordinate naturali (ξ, η, ζ), il cui valore varia nell'intervallo [-1,+1]. Le coordinate cartesiane (x, y, z) sono relazionate alle coordinate naturali per mezzo delle funzioni di base $N_i^{(e)}$:

$$x^{(e)} = \sum_{i=1}^{8} N_i^{(e)}(\xi,\eta,\zeta)x_i^{(e)} \tag{2.38}$$

$$y^{(e)} = \sum_{i=1}^{8} N_i^{(e)}(\xi,\eta,\zeta)y_i^{(e)} \tag{2.39}$$

$$z^{(e)} = \sum_{i=1}^{8} N_i^{(e)}(\xi,\eta,\zeta)z_i^{(e)} \tag{2.40}$$

in cui (x_i, y_i, z_i) sono le coordinate del nodo i-esimo. Le componenti del campo di spostamento, soluzione del problema elastico, sono definite come:

$$\begin{aligned} u^{(e)}(x,y,z) &= \sum_{i=1}^{8} N_i^{(e)}(\xi,\eta,\zeta)u_i^{(e)} \\ v^{(e)}(x,y,z) &= \sum_{i=1}^{8} N_i^{(e)}(\xi,\eta,\zeta)v_i^{(e)} \\ w^{(e)}(x,y,z) &= \sum_{i=1}^{8} N_i^{(e)}(\xi,\eta,\zeta)w_i^{(e)} \end{aligned} \tag{2.41}$$

Riscrivendo le componenti di spostamento in forma matriciale, si ottiene:

$$\begin{bmatrix} u^{(e)} \\ v^{(e)} \\ w^{(e)} \end{bmatrix} = \sum_{i=1}^{8} \begin{bmatrix} N_i^{(e)} & 0 & 0 \\ 0 & N_i^{(e)} & 0 \\ 0 & 0 & N_i^{(e)} \end{bmatrix} \begin{bmatrix} u_i^{(e)} \\ v_i^{(e)} \\ w_i^{(e)} \end{bmatrix} \tag{2.42}$$

La *matrice delle funzioni di base* ha quindi dimensioni [3x24]. In generale, le funzioni di base di un elemento esaedrico trilineare risultano:

$$N_i^{(e)}(\xi,\eta,\zeta) = \frac{1}{8}(1+\xi_i\xi)(1+\eta_i\eta)(1+\zeta_i\zeta) \quad con\ i = 1,..,8 \tag{2.43}$$

in cui le coordinate caratterizzate dal pedice i sono le coordinate dell'i-esimo nodo dell'elemento. Nel sistema di rifermento locale, per un elemento i cui vertici sono numerati come in

figura 2.2, le funzioni di base relative a ciascun nodo sono:

$$\begin{aligned}
N_1^{(e)} &= \frac{1}{8}(1-\xi)(1-\eta)(1-\zeta) \\
N_2^{(e)} &= \frac{1}{8}(1+\xi)(1-\eta)(1-\zeta) \\
N_3^{(e)} &= \frac{1}{8}(1+\xi)(1+\eta)(1-\zeta) \\
N_4^{(e)} &= \frac{1}{8}(1-\xi)(1+\eta)(1-\zeta) \\
N_5^{(e)} &= \frac{1}{8}(1-\xi)(1-\eta)(1+\zeta) \\
N_6^{(e)} &= \frac{1}{8}(1+\xi)(1-\eta)(1+\zeta) \\
N_7^{(e)} &= \frac{1}{8}(1+\xi)(1+\eta)(1+\zeta) \\
N_8^{(e)} &= \frac{1}{8}(1-\xi)(1+\eta)(1+\zeta)
\end{aligned} \tag{2.44}$$

Queste ultime, essendo polinomi a supporto locale, devono soddisfare due condizioni:

$$N_i^{(e)}(\xi_j, \eta_j, \zeta_j) = \begin{cases} 1 \; se \; i = j \\ 0 \; se \; i \neq j \end{cases} \tag{2.45}$$

$$\sum_{i=1}^{8} N_i^{(e)}(\xi, \eta, \zeta) = 1 \tag{2.46}$$

Le derivate delle funzioni di base sono ottenibili mediante la regola di derivazione a catena,

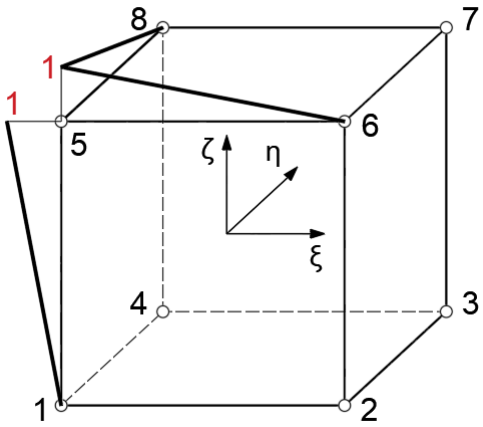


Figura 2.3: Funzioni di base per il nodo 5 che soddisfano la condizione (2.45).

per la quale:

$$\begin{aligned}
\frac{\partial N_i^{(e)}}{\partial x} &= \frac{\partial N_i^{(e)}}{\partial \xi}\frac{\partial \xi}{\partial x} + \frac{\partial N_i^{(e)}}{\partial \eta}\frac{\partial \eta}{\partial x} + \frac{\partial N_i^{(e)}}{\partial \zeta}\frac{\partial \zeta}{\partial x} \\
\frac{\partial N_i^{(e)}}{\partial y} &= \frac{\partial N_i^{(e)}}{\partial \xi}\frac{\partial \xi}{\partial y} + \frac{\partial N_i^{(e)}}{\partial \eta}\frac{\partial \eta}{\partial y} + \frac{\partial N_i^{(e)}}{\partial \zeta}\frac{\partial \zeta}{\partial y} \\
\frac{\partial N_i^{(e)}}{\partial z} &= \frac{\partial N_i^{(e)}}{\partial \xi}\frac{\partial \xi}{\partial z} + \frac{\partial N_i^{(e)}}{\partial \eta}\frac{\partial \eta}{\partial z} + \frac{\partial N_i^{(e)}}{\partial \zeta}\frac{\partial \zeta}{\partial z}
\end{aligned} \tag{2.47}$$

in cui *i=1,2,..,8* indica l'indice del nodo dell'elemento. Le equazioni appena enunciate si possono riscrivere in forma matriciale come segue:

$$\begin{bmatrix} \frac{\partial N_i^{(e)}}{\partial \xi} \\ \frac{\partial N_i^{(e)}}{\partial \eta} \\ \frac{\partial N_i^{(e)}}{\partial \zeta} \end{bmatrix} = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} & \frac{\partial z}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} & \frac{\partial z}{\partial \eta} \\ \frac{\partial x}{\partial \zeta} & \frac{\partial y}{\partial \zeta} & \frac{\partial z}{\partial \zeta} \end{bmatrix} \begin{bmatrix} \frac{\partial N_i^{(e)}}{\partial x} \\ \frac{\partial N_i^{(e)}}{\partial y} \\ \frac{\partial N_i^{(e)}}{\partial z} \end{bmatrix} = J^{(e)} \frac{\partial N_i^{(e)}}{\partial x} \tag{2.48}$$

La matrice $J^{(e)}$ è la *matrice jacobiana* dell'elemento, calcolabile come:

$$J^{(e)} = \sum_{i=1}^{8} \begin{bmatrix} \frac{\partial N_i^{(e)}}{\partial \xi} x_i & \frac{\partial N_i^{(e)}}{\partial \xi} y_i & \frac{\partial N_i^{(e)}}{\partial \xi} z_i \\ \frac{\partial N_i^{(e)}}{\partial \eta} x_i & \frac{\partial N_i^{(e)}}{\partial \eta} y_i & \frac{\partial N_i^{(e)}}{\partial \eta} z_i \\ \frac{\partial N_i^{(e)}}{\partial \zeta} x_i & \frac{\partial N_i^{(e)}}{\partial \zeta} y_i & \frac{\partial N_i^{(e)}}{\partial \zeta} z_i \end{bmatrix} \tag{2.49}$$

Di conseguenza, una volta calcolata la matrice jacobiana, le derivate delle funzioni di base sono ottenibili come:

$$\begin{bmatrix} \frac{\partial N_i^{(e)}}{\partial x} \\ \frac{\partial N_i^{(e)}}{\partial y} \\ \frac{\partial N_i^{(e)}}{\partial z} \end{bmatrix} = \left[J^{(e)}\right]^{-1} \begin{bmatrix} \frac{\partial N_i^{(e)}}{\partial \xi} \\ \frac{\partial N_i^{(e)}}{\partial \eta} \\ \frac{\partial N_i^{(e)}}{\partial \zeta} \end{bmatrix} \tag{2.50}$$

Si calcola, infine, la *matrice delle derivate delle funzioni di base*, di dimensioni [6x24]:

$$B^{(e)} = [B_1^{(e)}\ B_2^{(e)}\ B_3^{(e)}\ B_4^{(e)}\ B_5^{(e)}\ B_6^{(e)}\ B_7^{(e)}\ B_8^{(e)}] \tag{2.51}$$

La matrice delle derivate riferita al nodo i-esimo, con *i=1,2,..,8*, è pari a:

$$B_i^{(e)}(\xi,\eta,\zeta) = \begin{bmatrix} b_i & 0 & 0 \\ 0 & c_i & 0 \\ 0 & 0 & d_i \\ c_i & b_i & 0 \\ d_i & 0 & b_i \\ 0 & d_i & c_i \end{bmatrix} \tag{2.52}$$

in cui le componenti b_i, c_i, d_i si ricavano come:

$$\begin{bmatrix} b_i \\ c_i \\ d_i \end{bmatrix} = \sum_{k=1}^{3} \begin{bmatrix} \bar{J}_{1k}^{(e)} \\ \bar{J}_{2k}^{(e)} \\ \bar{J}_{3k}^{(e)} \end{bmatrix} \frac{\partial N_i^{(e)}}{\partial \xi_k} \tag{2.53}$$

in cui $\bar{J}_{ij}^{(e)}$ rappresenta il termine *ij* dell'inversa della matrice jacobiana e $\xi_1 = \xi$, $\xi_2 = \eta$, $\xi_3 = \zeta$. La matrice di rigidezza locale si ricava dall'equazione (2.25) e, data la formulazione *tri-lineare* dell'elemento esaedrico, richiede un processo di integrazione. Si impiega il metodo della quadratura di Gauss a due punti riferita all'elemento cubico normalizzato.

$$H^{(e)} = \int_{\Omega^{(e)}} B^{(e),T} D^{(e)} B^{(e)} \; det(J^{(e)}) \; d\Omega \tag{2.54}$$

La matrice di rigidezza valutata negli 8 punti Gauss di coordinate (ξ_s, η_t, ζ_p) si calcola come:

$$H^{(e)} = \sum_{i=1}^{8} B^{(e),T}(\xi_{s,i}, \eta_{t,i}, \zeta_{p,i}) D^{(e)} B^{(e)}(\xi_{s,i}, \eta_{t,i}, \zeta_{p,i}) \mid J^{(e)}(\xi_{s,i}, \eta_{t,i}, \zeta_{p,i}) \mid W_s W_t W_p \tag{2.55}$$

Le coordinate e il valore dei pesi di Gauss sono quelli riportati in tabella (2.1), relativamente al secondo ordine di quadratura (n=2). La matrice di rigidezza locale dell'esaedro trilineare ha dimensione [24x24].

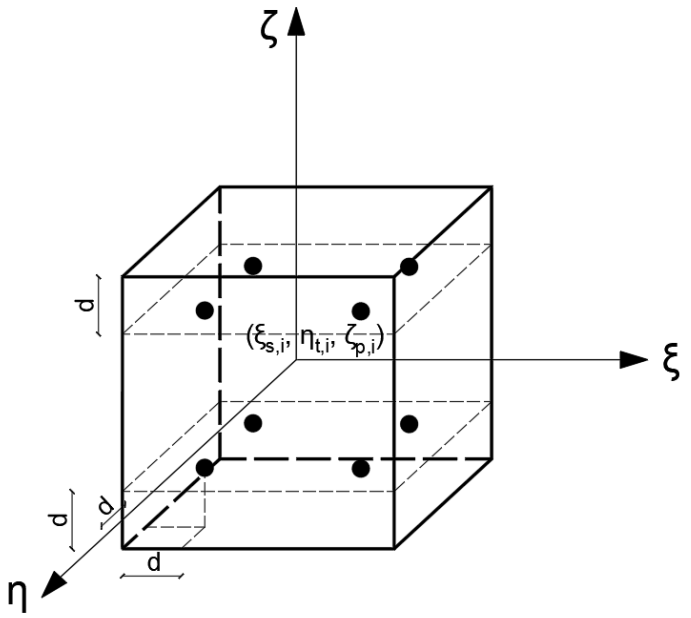


Figura 2.4: Punti Gauss per l'integrazione esatta di un esaedro trilineare ($d = \pm 1/\sqrt{3}$).

n	$\mathbf{x_i}$	$\mathbf{W_i}$
1	0.0	2.0
2	$\pm$0.5773502692	1.0
3	$\pm$0.774596697	0.5555555556
	0.0	0.8888888889
4	$\pm$0.8611363116	0.3478548451
	$\pm$0.3399810436	0.6521451549

Tabella 2.1: Tabella dei punti e dei pesi per le formule di quadratura di Gauss.

Quadratura di Gauss

La matrice di rigidezza dell'elemento esaedrico trilineare è stata ricavata tramite la tecnica di quadratura di Gauss. In generale, il concetto su cui si basano le tecniche di integrazione è quello di sostituire alla funzione integranda un polinomio di Lagrange, all'interno dell'intervallo di integrazione [a,b]. Si ipotizzi di voler calcolare il valore del seguente integrale nell'intervallo [+1,-1]:

$$I = \int_{-1}^{+1} f(x)\,dx \tag{2.56}$$

Il metodo di quadratura di Gauss approssima il calcolo dell'integrale con la sommatoria dei valori della funzione integranda calcolata in determinati punti di appoggio e moltiplicati per un peso predefinito:

$$I \simeq \sum_{i=1}^{n} f(x_i) W_i \tag{2.57}$$

in cui n è l'ordine di quadratura, x_i è il punto di appoggio di Gauss e W_i è il peso corrispondente all'i-esimo punto d'appoggio. I punti di appoggio del metodo di Gauss sono *ottimali*, cioè tali da calcolare in maniera esatta il valore di un polinomio di integrazione di grado *2n-1*. Le coordinate di questi punti e il valore dei pesi sono riportati in tabella 2.1. Si noti che tutti i punti d'appoggio di Gauss sono compresi nell'intervallo [-1,+1], per questo motivo conviene operare un cambio di variabili e riscrivere le funzioni di base degli elementi finiti in termini di coordinate naturali (ξ, η, ζ), come fatto per l'elemento esaedrico trilineare.

2.4.3 Assemblaggio

Una volta calcolate tutte le matrici di rigidezza dei singoli elementi, occorre assemblare ciascun termine all'interno della matrice di rigidezza globale del sistema H, nella posizione corretta. La dimensione della matrice di rigidezza globale è il prodotto tra il numero di gradi di libertà assegnati a ciascun nodo e il numero totale di nodi della *mesh*. Negli elementi finiti tridimensionali, ciascun nodo è libero di traslare lungo i tre assi definiti dal sistema di riferimento (x, y, z), di conseguenza ciascun nodo possiede 3 gradi di libertà. La dimensione della

matrice globale coincide anche con la lunghezza del vettore dei termini noti **r** e del vettore soluzione **s**.

Il *processo di assemblaggio* della matrice H si esegue collegando a ciascun elemento della matrice locale la posizione corrispondente nella matrice globale e viene indicato con la seguente simbologia:

$$H = \sum_{e} H_{ij}^{(e)} \tag{2.58}$$

Considerando, ad esempio, un elemento tetraedrico lineare con vertici *1, 2, 3, 4*, il coefficiente locale ${h_{1,1}}^{(e)}$ che tiene conto della rigidezza associata al *primo nodo* dell'elemento *e*, andrà posizionato nel coefficiente diagonale globale h_{ii}. Il coefficiente ${h_{1,3}}^{(e)}$, che tiene conto dell'interazione tra il primo e il terzo nodo dell'elemento *e*, andrà invece posizionato in h_{ik}. Ad uno stesso nodo, possono essere associati più elementi finiti e, in questo caso, il termine globale della matrice di rigidezza sarà pari alla *somma* dei contributi locali, provenienti da elementi diversi. Inoltre, nel caso in cui due nodi della mesh non siano a contatto, il termine di rigidezza che tiene conto dell'interazione risulterà nullo. Per questo motivo, genericamente la matrice di rigidezza globale è una matrice sparsa, ovvero una matrice con molti termini pari a zero. È evidente che se il nodo *i* è a contatto con il nodo *j*, anche *j* sarà a contatto con *i*, e dunque la *topologia* della matrice H, ovvero l'ubicazione dei termini non nulli, sarà *simmetrica*.

Per chiarire i concetti appena esposti, si riporta di seguito l'assemblaggio della matrice H nel caso di una struttura tridimensionale, discretizzata con elementi tetraedrici lineari. Si denotano con h_{ij} i termini di H, in cui *i* indica il numero del nodo e *j* indica il numero del nodo in contatto. Sia assegnata una topologia degli elementi, di cui si riporta solo una parte:

e	i	j	k	m
1	5	7	8	16
2	5	1	6	12
3	8	4	5	19
4	4	2	5	18
5	4	3	2	13
6	4	8	9	14

Si ipotizzi di voler assemblare i contributi locali dell'elemento **3**, la cui topologia è data dalla successione dei nodi *8, 4, 5, 19*. I termini della matrice di rigidezza locale si collegano ai termini della matrice globale secondo il seguente schema:

	8	4	5	19
8	$h^{(3)}_{1,1}$	$h^{(3)}_{1,2}$	$h^{(3)}_{1,3}$	$h^{(3)}_{1,4}$
4	$h^{(3)}_{2,1}$	$h^{(3)}_{2,2}$	$h^{(3)}_{2,3}$	$h^{(3)}_{2,4}$
5	$h^{(3)}_{3,1}$	$h^{(3)}_{3,2}$	$h^{(3)}_{3,3}$	$h^{(3)}_{3,4}$
19	$h^{(3)}_{4,1}$	$h^{(3)}_{4,2}$	$h^{(3)}_{4,3}$	$h^{(3)}_{4,4}$

Il termine $h^{(3)}_{1,1}$ sarà il termine $h_{8,8}$ all'interno della matrice globale, il termine $h^{(3)}_{1,2}$, invece, sarà il termine $h_{8,4}$ e così via. Nel caso in cui ci siano più termini locali che concorrono al medesimo nodo, i loro contributi vengono sommati. Ad esempio, per l'assemblaggio del termine diagonale $h_{5,5}$ si sommano tutti i contributi locali derivanti da tutti gli elementi afferenti al *nodo 5*, come ad esempio gli elementi *1*, *2*, *3* e *4*. Il termine diagonale della matrice di rigidezza, in riferimento al nodo 5, sarà la somma di più contributi tra cui $h^{(1)}_{1,1}, h^{(2)}_{1,1}, h^{(3)}_{3,3}, h^{(4)}_{3,3}$. Infine, in base alle considerazioni esposte sopra, i termini *non nulli* della matrice globale saranno tutti i termini diagonali (perché ogni nodo è in contatto con sé stesso) e tutti i termini che tengono conto dell'interazione tra i nodi dell'elemento 3 e i nodi degli altri elementi.

Per quanto riguarda la risoluzione del problema elastico, una volta assemblata la matrice globale e il vettore dei termini noti, si risolve il sistema (2.26).

Capitolo 3

Modelli costitutivi

Per modellare il comportamento meccanico di un qualsiasi mezzo è necessario identificare la relazione che lega le tensioni alle deformazioni, ovvero il modello costitutivo del materiale. I modelli costitutivi sono dei modelli teorici rappresentativi di particolari comportamenti ideali (elastico, plastico, lineare, ecc.) che un materiale può sviluppare. A seconda del tipo di terreno che si intende modellare (sabbia, argilla, limo ecc.) e del tipo di comportamento atteso, si possono adottare diversi modelli costitutivi, in relazione anche al numero di parametri meccanici richiesti in ciascun legame. In generale, infatti, la formulazione del legame tensioni-deformazioni necessita l'introduzione di una *matrice costitutiva D*, di ordine 6x6 (nel caso di un problema tridimensionale), le cui componenti sono funzione dei parametri meccanici del materiale. Se i parametri meccanici del mezzo sono costanti in ogni punto, il materiale si può definire omogeneo. Il modello costitutivo di un materiale si definisce genericamente come:

$$d\sigma = D(\sigma)\, d\varepsilon \tag{3.1}$$

in cui $d\sigma$ è una variazione infinitesima del tensore delle tensioni introdotto in (2.7) e $d\varepsilon$ è la corrispondente variazione del tensore delle deformazioni presentato in (2.4).

Volendo riportare la relazione appena enunciata in un grafico, si ottiene il diagramma in figura 3.1. Il primo tratto O-A è caratterizzato da un comportamento *elastico* e *lineare*. La tensione è infatti direttamente proporzionale alla deformazione e le deformazioni che si sviluppano in questo tratto sono di tipo elastico, ovvero completamente recuperabili una volta terminato il processo di carico. Nel tratto A-B il comportamento è invece *elastico* e *non lineare*, la deformazione aumenta in proporzione non lineare all'aumentare dei carichi agenti esternamente. Nel punto B il valore del carico esterno genera una tensione limite all'interno del mezzo, detta *tensione di snervamento*, oltre la quale il comportamento del materiale risulta *plastico* cioè le deformazioni che si sviluppano oltre questo limite sono irreversibili. Nel presente capitolo si riportano tutti i modelli costitutivi implementati finora in GReS.

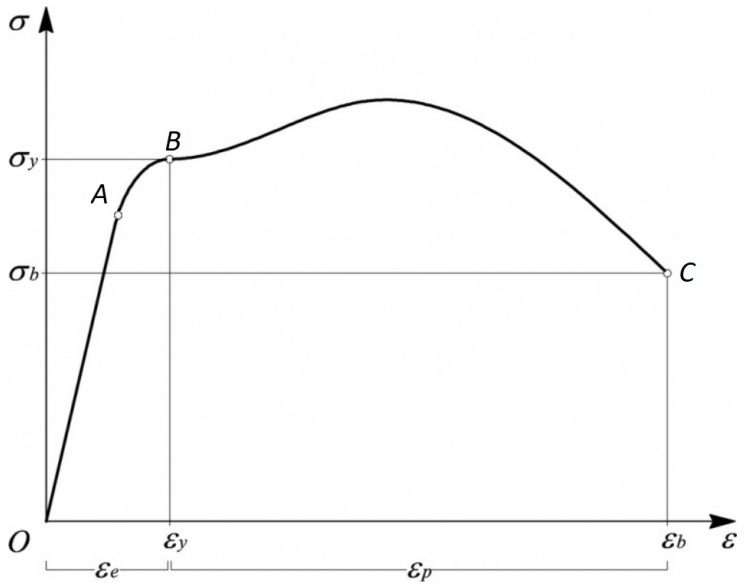


Figura 3.1: Diagramma del legame tensioni-deformazioni durante una prova di carico.

3.1 Modelli elastici

Si consideri un dominio a forma di parallelepipedo con sezione retta A, di altezza iniziale h_i, soggetto ad una forza di compressione assiale P. L'abbassamento del parallelepipedo e la forza applicata sono tra loro direttamente proporzionali e, in un materiale elastico, la costante di proporzionalità è il modulo di elasticità E. L'abbassamento del parallelepipedo è pari a:

$$\Delta h = h_f - h_i = \frac{P}{E} h_i \tag{3.2}$$

in cui h_f è l'altezza al termine del processo di carico. Si assume che l'asse x sia l'asse longitudinale della parallelepipedo. La deformazione è una grandezza adimensionale, calcolabile come il rapporto tra l'abbassamento e l'altezza iniziale:

$$\varepsilon_x = \frac{h_f - h_i}{h_i} = \frac{P}{EA} \tag{3.3}$$

Mentre la tensione agente è data da:

$$\sigma_x = \frac{P}{A} \tag{3.4}$$

Se a partire dall'equazione 3.3, si esplicita la trazione P e si sostituisce all'interno dell'equazione 3.4, si ottiene:

$$\sigma_x = E\varepsilon_x \tag{3.5}$$

che rappresenta il legame elastico lineare per un caso di trazione monoassiale.

Le considerazioni esposte per la prova di compressione monoassiale sono estendibili anche al continuo tridimensionale, in cui il tensore delle tensioni è proporzionale al tensore delle deformazioni per mezzo della matrice costitutiva D, le cui componenti sono costanti,

nel caso di materiale elastico lineare, oppure dipendenti dallo stato tensionale, in caso di materiale elastico non lineare. La matrice D è simmetrica e, nel caso generale di materiale anisotropo, contiene le 21 costanti elastiche del materiale. Tuttavia, in presenza di simmetrie parziali all'interno del mezzo, il numero di costanti elastiche necessarie alla definizione del legame si riduce. La riduzione delle costanti comporta generalmente anche una riduzione delle prove sperimentali necessarie alla definizione dei parametri meccanici del materiale.

3.1.1 Modello elastico lineare isotropo

Il legame *elastico lineare isotropo* si basa sulla legge di Hooke e necessita di appena *2 parametri meccanici* per il calcolo della matrice costitutiva. L'isotropia, infatti, definisce un gruppo di materiali il cui comportamento non presenta direzioni preferenziali.
I due parametri indipendenti che governano il modello sono il modulo di elasticità longitudinale (o mudulo di Young) E e il coefficiente di contrazione trasversale (o di Poisson) ν. La matrice costitutiva del legame elastico lineare isotropo risulta:

$$D = \frac{E}{(1+\nu)(1-2\nu)} \begin{bmatrix} (1-\nu) & \nu & \nu & 0 & 0 & 0 \\ \nu & (1-\nu) & \nu & 0 & 0 & 0 \\ \nu & \nu & (1-\nu) & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1-2\nu}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1-2\nu}{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1-2\nu}{2} \end{bmatrix} \tag{3.6}$$

in cui G è il modulo di elasticità tangenziale ed è pari a $E/2(1+\nu)$. Una volta noti i due parametri meccanici del materiale, è definito il legame tra componenti di tensione e deformazione. L'impiego del modello elastico lineare isotropo è opportuno nel caso in cui le deformazioni sviluppate durante l'analisi siano di tipo elastico, ovvero recuperabili al termine del processo di scarico. Inoltre, il materiale di cui è costituito il mezzo deve essere isotropo e omogeneo. In caso contrario, i risultati ottenuti potrebbero essere poco significativi del reale comportamento del terreno.

3.1.2 Modello elastico lineare ortotropo

I materiali ortotropi presentano tre piani di simmetria tra loro ortogonali. La condizione di ortogonalità tra gli assi principali garantisce un disaccoppiamento tra le tensioni normali agenti nelle direzioni principali (σ_{ii}) e gli scorrimenti (τ_{ij}). In generale, indicando con x, y, e z le direzioni del sistema di riferimento, in caso di simmetria rispetto ai piani (x,y), (y,z) e (x,z) la matrice di rigidezza conta 18 termini nulli, che tengono conto della mancanza di associazione tra tensioni normali e deformazioni di taglio.

In particolare, un materiale si definisce *trasversalmente isotropo* se esiste una direzione preferenziale ed è isotropo in un piano perpendicolare a questa direzione. Un materiale elastico lineare trasversalmente isotropo è definito da 4 parametri meccanici, da cui dipendono le *5*

costanti elastiche. Nell'ipotesi in cui l'asse z è l'asse di simmetria e il piano (x,y) è il piano di isotropia, le costanti del legame costitutivo risultano:

$$E_x = E_y = E \tag{3.7}$$

$$E_z = E_Z = \frac{E}{\lambda} \tag{3.8}$$

$$\nu_{zx} = \nu_{zy} = \nu_Z \tag{3.9}$$

$$\nu_{xy} = \nu_H \tag{3.10}$$

$$G = \frac{E}{2(1+\nu_Z)} \tag{3.11}$$

in cui:

- E rappresenta il modulo di elasticità longitudinale del materiale nel piano isotropo (perpendicolare all'asse di simmetria);
- E_Z rappresenta il modulo di elasticità longitudinale lungo l'asse di simmetria, ed è pari a E/λ;
- ν_H rappresenta il coefficiente di Poisson nel piano isotropo;
- ν_Z rappresenta il coefficiente di Poisson quando la tensione è applicata lungo l'asse di simmetria;
- G è il modulo di elasticità tangenziale funzione delle altre costanti elastiche $G = E/2(1+\nu_Z)$.

La matrice costitutiva per un materiale trasversalmente isotropo risulta:

$$D = \frac{E}{(\lambda - \lambda\nu_H - 2\nu_Z^2)} \begin{bmatrix} \frac{(\lambda-\nu_Z^2)}{(1+\nu_H)} & \frac{(\lambda\nu_H+\nu_Z^2)}{(1+\nu_H)} & \nu_Z & 0 & 0 & 0 \\ \frac{(\lambda\nu_H+\nu_Z^2)}{(1+\nu_H)} & \frac{(\lambda-\nu_Z^2)}{(1+\nu_H)} & \nu_Z & 0 & 0 & 0 \\ \nu_Z & \nu_Z & (1-\nu_H) & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{(\lambda-\lambda\nu_H-2\nu_Z^2)}{2(1+\nu_Z)} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{(\lambda-\lambda\nu_H-2\nu_Z^2)}{2(1+\nu_Z)} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{(\lambda-\lambda\nu_H-2\nu_Z^2)}{2(1+\nu_H)} \end{bmatrix} \tag{3.12}$$

3.1.3 Modello ipoelastico

Il modello costitutivo *ipoelastico* è un modello elastico isotropo *non lineare*, in cui la compressibilità verticale non è costante poiché dipende dalla tensione verticale del mezzo. Questo modello si descrive in termini di variazioni di tensione variazioni di deformazione come segue:

$$d\sigma = D(\sigma)\, d\varepsilon \tag{3.13}$$

in cui D è la matrice costitutiva del materiale:

$$D(\sigma) = C_M^{-1}(\sigma) \begin{bmatrix} 1 & \frac{\nu}{1-\nu} & \frac{\nu}{1-\nu} & 0 & 0 & 0 \\ \frac{\nu}{1-\nu} & 1 & \frac{\nu}{1-\nu} & 0 & 0 & 0 \\ \frac{\nu}{1-\nu} & \frac{\nu}{1-\nu} & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1-2\nu}{2(1-\nu)} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1-2\nu}{2(1-\nu)} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1-2\nu}{2(1-\nu)} \end{bmatrix} \tag{3.14}$$

La compressibilità verticale del materiale C_M in condizioni edometriche dipende dalla tensione verticale e, dato che il terreno si comporta in modo differente a seconda che si trovi in fase di compressione vergine o in fase di scarico/ricarico, la sua formulazione varia in relazione alla tensione di preconsolidazione del terreno σ_P:

- Se $\sigma_z \leq \sigma_P$, ovvero il terreno è in fase di compressione vergine:

$$C_M^I(\sigma) = a \mid \sigma_z \mid^b \tag{3.15}$$

- Se $\sigma_z > \sigma_P$, ovvero il terreno è in fase di scarico/ricarico:

$$C_M^{II}(\sigma) = a' \mid \sigma_z \mid^{b'} \tag{3.16}$$

in cui a e b sono parametri costanti e $\sigma_z \leq 0$. La rappresentazione grafica di C_M è riportata in figura 3.2.

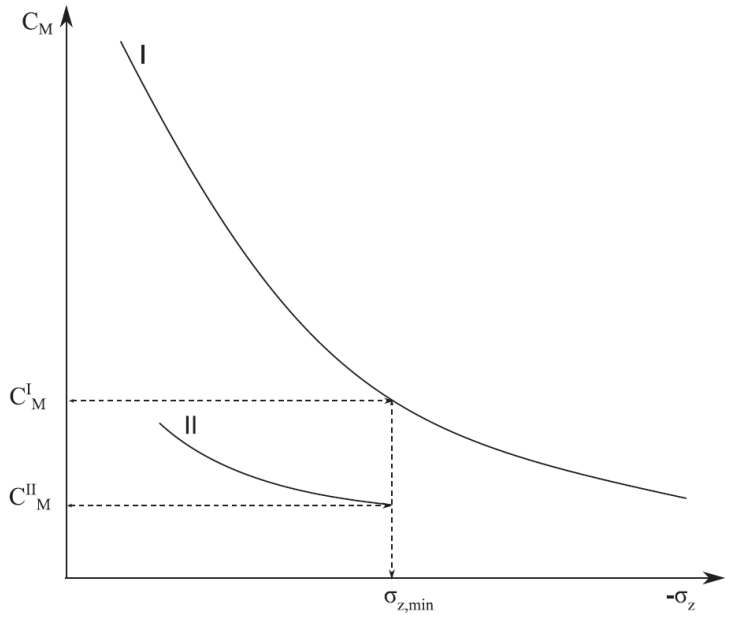


Figura 3.2: Andamento della compressibilità al variare della tensione verticale in un legame ipoelastico [Spiezia et al., 2017].

Capitolo 4

Struttura del codice

4.1 Programmazione a classi in Matlab

Come anticipato nell'introduzione, si è scelto di realizzare GReS utilizzando la programmazione a classi in Matlab che favorisce la *modularità* del codice. Si è scelto di utilizzare Matlab poiché è un linguaggio di programmazione di alto livello, molto diffuso e con un basso costo di accesso. Risulta di più semplice utilizzo rispetto ai linguaggi di basso livello poiché è orientato verso l'utente, piuttosto che a svolgere le operazioni computazionali logico-matematiche o di controllo del computer. Tuttavia, permette di richiamare routine sviluppate in linguaggi di basso livello, che risultano più efficienti, senza quindi perdere troppa efficienza computazionale.

Con il termine *programmazione a classi* si intende quel paradigma di programmazione per il quale i dati di un problema vengono incapsulati all'interno di una o più classi, e possono interagire solo ed esclusivamente con i metodi contenuti nella classe di appartenenza. Poiché non c'è connessione tra oggetti e metodi di classi diverse, è possibile che l'implementazione del codice avvenga da parte di più programmatori, anche contemporaneamente. Inoltre, l'organizzazione del codice sotto forma di classi agevola i processi di *aggiornamento* e/o *integrazione*. È bene precisare anche che il concetto di programmazione a classi e programmazione ad oggetti non coincidono. Le classi rappresentano, infatti, un modello per mezzo del quale si possono creare più oggetti, caratterizzati dai medesimi metodi e dalle medesime proprietà. Inoltre, la programmazione a classi è implementata solo in C++, Java e Matlab, mentre la programmazione ad oggetti si può impiegare in qualsiasi linguaggio di programmazione.

Le *classi* sono dei modelli astratti tramite i quali creare *oggetti* dotati di *proprietà*, ovvero dati, e di *metodi*, ovvero procedure che operano sui dati dell'oggetto. La struttura dati di una classe è contenuta in un file con estensione *.m*, il cui nome coincide col nome della classe stessa.

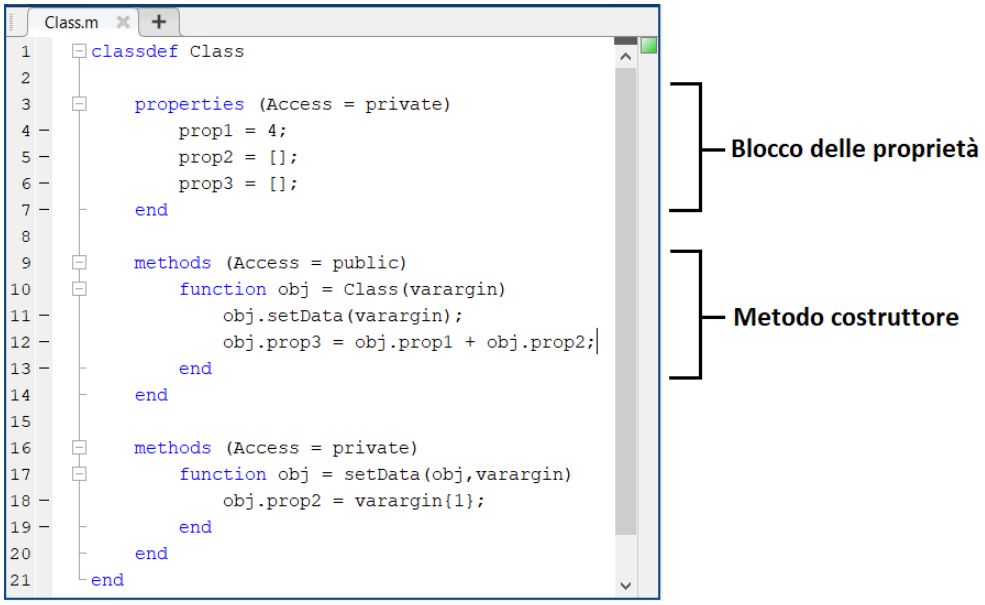


Figura 4.1: Generica struttura dati di una classe in ambiente Matlab.

4.1.1 Proprietà di una classe

Le proprietà all'interno di una classe raggruppano i *dati* dell'oggetto generato dalla classe stessa. Si possono definire più blocchi di proprietà all'interno di una classe, nel caso in cui sia necessario diversificare il comportamento di alcuni dati. Le proprietà, infatti, sono dotate di *attributi* che ne modificano determinati aspetti. In GReS, ad esempio, si è scelto di impedire l'accesso all'utente a determinate proprietà, specificando che l'attributo *Access* fosse pari alla parola chiave *private*. Le proprietà ad accesso libero sono invece contraddistinte dall'istruzione *Access* = *public*. Per specificare l'attributo di un blocco di proprietà è sufficiente inserire il nome dell'attributo e il suo valore nella stessa riga in cui viene inizializzato il blocco delle proprietà (figura 4.1). I valori di ciascun dato all'interno del blocco delle proprietà possono essere assegnati di default oppure possono essere assegnati a partire dagli output generati dal metodo costruttore o dagli altri metodi della classe, come mostrato in figura 4.1.

La programmazione a classi in Matlab garantisce, dunque, un ampio controllo su diversi aspetti delle proprietà e, di conseguenza, dei dati dell'oggetto.

4.1.2 Metodi di una classe

In generale, i metodi di una classe raggruppano le *function* che possono operare sui dati della classe stessa. Esiste inoltre un particolare tipo di metodo, detto *metodo costruttore*, che porta lo stesso nome della classe e viene di solito utilizzato per assegnare i valori delle proprietà dell'oggetto. Uno dei grandi vantaggi della programmazione ad oggetti è dato dalla possibilità di implementare più function all'interno di una classe, che possono essere utilizzate o meno a seconda degli output richiesti dal problema. Inoltre, come avviene per le proprietà, è possibile personalizzare i metodi con diverse tipologie di attributi, come in figura 4.1.

4.2 GReS: classi implementate

Le classi attualmente implementate in GReS rappresentano il punto di partenza per la realizzazione di uno strumento di simulazione più completo ed efficiente.

Il paradigma di programmazione a classi ha permesso di gettare le basi di un codice che risolve il problema dell'equilibrio elastico di un dominio tridimensionale. Le classi e le function implementate finora all'interno del codice si occupano della definizione della griglia, dell'assemblaggio della matrice di rigidezza globale e dell'imposizione delle condizioni al contorno. In particolare:

- la lettura dei dati relativi al dominio discretizzato quali, ad esempio, il numero totale di nodi della mesh e le loro coordinate, la tipologia di elementi finiti impiegati e la topologia di ciascuno di essi, è gestita dalla classe *Mesh*;
- l'identificazione del modello costitutivo dei materiali è affidato alla classe *Materials*, mentre il calcolo della matrice di rigidezza di ciascun materiale è gestito dalle classi dei legami costitutivi (attualmente *Elastic*, *TransvElastic* e *HypoElastic*). Sono inoltre state create due classi per la gestione dei parametri di un problema di flusso, *Fluid* e *PorousRock*;
- l'identificazione della tipologia di condizioni al contorno è affidata alla classe *Boundaries*, mentre la creazione degli oggetti che gestiscono l'entità e i gradi di libertà vincolati, sono affidate alla classe *NodeBC*;
- l'identificazione della tipologia di elemento finito impiegato nella discretizzazione del dominio è affidato alla classe *Elements*, la quale rimanda alle classi *Tetrahedron* e *Hexahedron* per il calcolo delle proprietà degli elementi, quali ad esempio il volume o le funzioni di base. Al momento GReS è in grado di gestire solamente tetraedri lineari ed esaedri trilineari.

Oltre alle classi sopracitate, sono state implementate due funzioni per l'assemblaggio della matrice di rigidezza globale e una funzione per l'imposizione delle condizioni al contorno.

Si elencano tutte le classi implementate in GReS e, a seguire, si riporta la descrizione più dettagliata dei metodi e delle proprietà di ciascuna classe, corredata dalla lista di dati di input richiesti per la creazione di ciascun oggetto. Le classi sono:

- classe Mesh;
- classe Materials:
  - classe Elastic;
  - classe TransvElastic;
  - classe HypoElastic;
  - classe Fluid;

  - classe PorousRock;

- classe Boundaries:

  - classe NodeBC;

- classe Elements:

  - classe Tetrahedron;
  - classe Hexahedron.

È bene sottolineare che non sussiste alcuna gerarchia tra le classi implementate. Le tre function che operano l'assemblaggio della matrice di rigidezza e l'imposizione delle condizioni al contorno sono *assemblyK*, *assemblyKGauss* e *imposeBC*, rispettivamente. Si sono implementate due function per l'assemblaggio della matrice di rigidezza a seconda che sia necessario o meno l'integrazione della matrice tramite la tecnica della quadratura di Gauss.

4.2.1 Dominio discretizzato

Il metodo degli elementi finiti prevede un processo di discretizzazione del dominio, che avviene attraverso la definizione di una griglia, detta *mesh*, composta da elementi finiti tridimensionali. La geometria degli elementi è definita da una determinata sequenza di nodi, ovvero entità monodimensionali la cui ubicazione nello spazio è individuata da un set di coordinate. Il primo passo per avviare una simulazione è, dunque, quello di generare la mesh del modello e memorizzarne le informazioni.

Classe *Mesh*

La lettura del file di input in cui è definita la mesh del modello discretizzato, è gestito dalla classe *Mesh*. Per l'avvio di una simulazione in GReS è necessario un file di input di tipo *.msh*, ovvero un file generato con il programma Gmsh (figura 4.2).

Le prime righe del file di input definiscono il formato della mesh mediante una serie di informazioni che comprendono il tipo di versione del file, il formato floating point adottato nel file, il numero e il nome delle regioni fisiche. Dopodiché sono elencati il numero e le coordinate di ciascun nodo e, per quanto riguarda gli elementi, il numero e la topologia.

La classe *Mesh*, inoltre, calcola le coordinate del centroide delle superfici e/o degli elementi finiti. All'interno delle proprietà vengono inizializzati:

- **nDim**: la dimensione della mesh (3D-2D);
- **nNodes**: il numero totale di nodi;
- **nCells**: il numero totale di elementi tridimensionali;
- **nSurfaces**: il numero totale di elementi bidimensionali;

```
$MeshFormat
2.2 0 8
$EndMeshFormat
$PhysicalNames
1
3 1 "reservoir"
$EndPhysicalNames
$Nodes
12
1 0 0 0
2 50 0 0
3 50 70 0
4 0 70 0
5 0 0 60
6 50 0 60
7 50 70 60
8 0 70 60
9 50 100 0
10 0 100 0
11 50 100 60
12 0 100 60
$EndNodes
$Elements
2
1 5 2 1 1 2 3 4 1 6 7 8 5
2 5 2 1 1 3 9 10 4 7 11 12 8
$EndElements
```

Figura 4.2: Esempio di file input per la classe *Mesh*.

- **coordinates**: la matrice delle coordinate dei nodi;
- **cells**: la matrice della topologia degli elementi tridimensionali;
- **cellTag**: il vettore contentente l'indice di materiale relativo a ciascun elemento tridimensionale;
- **cellNumVerts**: il numero di nodi di ciascun elemento tridimensionale;
- **surfaces**: la matrice della topologia degli elementi bidimensionali;
- **surfaceTag**: il vettore contentente l'indice di materiale relativo a ciascun elemento bidimensionale;
- **surfaceNumVerts**: il numero di nodi di ciascun elemento bidimensionale;
- **cellRegions**: la regione fisica a cui è associato ciascun elemento tridimensionale;
- **surfaceRegions**: la regione fisica a cui è associato ciascun elemento bidimensionale;
- **cellCentroid**: la matrice delle coordinate dei centroidi di ciascun elemento tridimensionale;
- **surfaceCentroid**: la matrice delle coordinate dei centroidi di ciascun elemento bidimensionale;
- **cellVTKType**: l'indice che identifica la tipologia di elemento tridimensionale sulla base della classificazione VTK;

- **surfaceVTKType**: l'indice che identifica la tipologia di elemento bidimensionale sulla base della classificazione VTK;
- **meshType**: la tipologia della mesh (strutturata o non strutturata).

Le proprietà appena elencate sono accessibili anche a funzioni non facenti parte della classe *Mesh*. Al contrario la proprietà ***typeMapping***, che contiene tutti gli indici della classificazione VTK, può essere utilizzata solo all'interno della classe. Per completezza, si riporta di seguito la classificazione del software opensource VTK che viene utilizzata anche all'interno di GReS.

Indice VTK	Elemento VTK
1	Vertice
2	Polivertice
3	Linea
4	Polilinea
5	Triangolo
6	Stinga di triangoli
7	Poligono
8	Pixel
9	Quadrangolo
10	Tetraedro
11	Voxel
12	Esaedro
13	Prisma triangolare
14	Piramide
15	Prisma pentagonale
16	Prisma esaedrico

Tabella 4.1: Tabella degli indici di elemento secondo la classificazione VTK.

La classe *Mesh* conta solo metodi di tipo *public*:

- la function *importGMSHmesh* è la funzione che assegna tutte le proprietà relative alla geometria e alla topologia della mesh che sono contenute nel file di input in figura 4.2;
- la function *finalize* richiama le function per il calcolo dei centroidi;
- le function *computeCellCentroid* e *computeSurfaceCentroid* calcolano le coordinate dei centroidi di ciascun elemento tridimensionale e di ciascuna superficie, rispettivamente;
- la function *findCellsOfRegion* e *findSurfaceOfRegion* permettono all'utente di controllare che un determinato elemento tridimensionale e le sue facce facciano parte di una delle regioni fisiche indicate nel file di input.

All'interno della function *mxImportGMSHmesh*, viene utilizzato un altro tipo di codifica per la tipologia di elemento finito, che deve coincidere con quella riportata nel file di input:

Indice	Tipo di elemento
1	Asta
2	Triangolo
3	Quadrangolo
4	Tetraedro
5	Esaedro
6	Prisma triangolare
7	Piramide

Tabella 4.2: Tabella degli indici di elemento all'interno della function *mxImportGMSHmesh*.

4.2.2 Modelli costitutivi

Sono state implementate 3 classi per la gestione dei legami costitutivi dei materiali e 2 classi per la gestione dei parametri necessari alla risoluzione di un modello di flusso.

Classe *Materials*

La classe *Materials* è la classe che si occupa dell'individuazione del tipo di legame costitutivo dei materiali, a partire da quanto specificato nel file di input. Le proprietà contano un solo parametro, *db*, e sono di tipo *public*, ovvero accessibili anche all'interno di altre classi e/o funzioni. Il parametro in questione è un oggetto *"Mappa"* che associa ogni *valore* al suo interno ad una *chiave*, che può essere di tipo numerico o testuale. Questa particolare struttura dati permette di memorizzare il codice specificato nel file di input che identifica il tipo di legame costitutivo del materiale, per poi valutare l'esistenza di una corrispondenza con il codice del materiale inserito dall'utente. La classe *Materials* conta metodi di tipo *public* e metodi di tipo *private*:

- la function *Materials* è il costruttore della classe che inizializza l'oggetto e richiama la function *readInputFile* per leggere e memorizzare i dati presenti nel file di input (figura 4.3);
- la function *getMaterial* utilizza l'oggetto *db* per cercare corrispondenza tra il tipo di legame costitutivo specificato nel file di input e il legame costitutivo definito dall'utente.

Il file di input per la creazione di un oggetto con la classe *Materials* deve quindi sempre includere:

- una stringa (*matName*) che identifica il tipo di modello costitutivo del materiale;

- una stringa (*matIdentifier*) che identifica il codice del modello costitutivo che viene memorizzato all'interno dell'oggetto *db*;
- i parametri meccanici del materiale, che variano a seconda del modello costitutivo assunto;
- la stringa *End* che indica la fine del set di dati di ciascun materiale.

```
1  Elastic                                  % material model
2  elas                                     % name
3  0.2e0      0.15e0                        % E, nu
4  End
5
6  HypoElastic                              % material model
7  hypoel                                   % name
8  0.2 1.50e0 0.10e0 1.0e 0.8e0 150        % nu, a, b, a1, b1, szmin
9  End
10
11 TransvElastic                            % material model
12 transvel                                 % name
13 100     150     0.3   0.4                % Ep, Ez, nu_p, nu_z
14 End
15
16 PorousRock                               % material model
17 rock                                     % name
18 0.5e-7     0.5e-7     0.5e-7   0.4      % permx, permy, permz, porosity
19 End
20
21 Fluid                                    % material model
22 fluid                                    % name
23 9.81e-3      1                           % specif weight, compressibility
24 End
```

Figura 4.3: Esempio di file input per la classe *Materials*.

Classe *Elastic*

La classe *Elastic* gestisce i materiali elastici lineari isotropi. Attualmente, legge e memorizza i parametri meccanici del materiale e calcola la matrice costitutiva locale. All'interno delle proprietà vengono inizializzati:

- **E**: modulo di elasticità longitudinale;
- ν: coefficiente di Poisson.

Le proprietà, in questo caso, sono accessibili solo ai metodi implementati all'interno della classe, perciò è stato specificato l'attributo *Access = private*. La classe *Elastic* conta metodi di tipo *public* e metodi di tipo *private*:

- la function *Elastic* è il costruttore della classe che inizializza l'oggetto e rimanda alla function *setMaterialParameters* per l'assegnazione di **E** e ν a partire dal valore riportato nel file di input (figura 4.3);
- la function *getStiffnessMatrix* calcola la matrice di rigidezza del legame costitutivo, come spiegato al paragrafo 3.1.1.

Classe *TransvElastic*

La classe *TransvElastic* gestisce i materiali elastici lineari trasversalmente isotropi. Attualmente, legge e memorizza i parametri meccanici del materiale e calcola la matrice costitutiva locale. All'interno delle proprietà (di tipo *private*) vengono inizializzati:

- **E**: modulo di elasticità longitudinale nel piano di simmetria;
- $\mathbf{E_Z}$: modulo di elasticità longitudinale lungo la direzione perpendicolare al piano di simmetria;
- $\nu_\mathbf{H}$: coefficiente di Poisson nel piano di simmetria;
- $\nu_\mathbf{Z}$: coefficiente di Poisson lungo la direzione perpendicolare al piano di simmetria.

La classe *TransvElastic* conta metodi di tipo *public* e metodi di tipo *private*, tra cui il costruttore della classe (function *TransvElastic*), le function per l'assegnazione dei parametri meccanici del materiale e per il calcolo della matrice di rigidezza, come spiegato al paragrafo 3.1.2.

Classe *HypoElastic*

La classe *HypoElastic* gestisce i materiali ipoelastici, caratterizzati da una compressibilità che varia in funzione della tensione nel mezzo. Attualmente, la classe legge e memorizza i parametri meccanici del materiale e calcola la matrice costitutiva locale. All'interno delle proprietà (di tipo *private*) vengono inizializzati:

- ν: coefficiente di Poisson;
- $\mathbf{a}$, $\mathbf{b}$, $\mathbf{a'}$, $\mathbf{b'}$: parametri per il calcolo della compressibilità che differiscono lungo il tratto di carico e lungo il tratto di scarico/ricarico;
- $\sigma_\mathbf{P}$: valore della tensione di preconsolidamento.

La classe *HypoElastic* conta metodi di tipo *public* e metodi di tipo *private* tra cui il costruttore della classe (function *HypoElastic*), le function per l'assegnazione dei parametri meccanici del materiale e per il calcolo della matrice di rigidezza, come spiegato nel paragrafo 3.1.3. La function *getCompressibility* calcola la compressibilità del materiale in base alla tensione verticale nel mezzo.

Classe *PorousRock*

La classe *PorousRock* gestisce i valori di permeabilità e porosità al materiale. All'interno delle proprietà (di tipo *private*) vengono inizializzati:

- $\mathbf{k_x}$: valore di permeabilità lungo la direzione x;

- $\mathbf{k_y}$: valore di permeabilità lungo la direzione y;
- $\mathbf{k_z}$: valore di permeabilità lungo la direzione z;
- **poro**: indice di porosità.

La classe *PorousRock* conta metodi di tipo *public* e metodi di tipo *private*, tra cui il costruttore della classe (function *PorousRock*) e la function per l'assegnazione delle proprietà dell'oggetto. La function *getPermeability* e la function *getPorosity* permettono all'utente di accedere ai valori di permeabilità e all'indice di porosità del mezzo, rispettivamente.

Classe *Fluid*

La classe *Fluid* gestisce i valori di peso specifico e compressibilità al fluido che permea il mezzo poroso. All'interno delle proprietà (di tipo *private*) vengono inizializzati:

- $\gamma_\mathbf{w}$: valore di peso specifico del fluido;
- β: valore di compressibilità del fluido.

La classe *PorousRock* conta metodi di tipo *public* e metodi di tipo *private*, tra cui il costruttore della classe (function *Fluid*) e la function per l'assegnazione delle proprietà dell'oggetto. La function *getWeight* e la function *getCompressibility* permettono all'utente di accedere al valore di peso specifico e di compressibilità del fluido, rispettivamente.

4.2.3 Condizioni al contorno

È stata implementata una classe per la gestione delle condizioni al contorno, per le condizioni assegnate direttamente sui nodi della mesh.

Classe *Boundaries*

La classe *Boundaries* è la classe che si occupa dell'individuazione del tipo di condizione al contorno, a partire da quanto specificato nel file di input. Le proprietà contano un solo parametro, *db*, e sono di tipo *private*. Il parametro in questione è sempre un oggetto *"Mappa"*, di cui si è parlato nella classe *Materials* che permette di memorizzare il codice per l'identificazione del tipo di condizione al contorno specificato nel file di input, per poi valutare l'esistenza di una corrispondenza con il codice inserito dall'utente. La classe *Boundaries* conta metodi di tipo *public* e metodi di tipo *private*:

- la function *Boundaries* è il costruttore della classe che inizializza l'oggetto e richiama la function *readInputFile* per leggere e memorizzare i dati presenti nel file di input (figura 4.4);
- la function *getBC* utilizza l'oggetto *db* per cercare corrispondenza tra il tipo di condizione al contorno specificata nel file di input e il tipo definito dall'utente.

Il file di input per la creazione di un oggetto con la classe *Boundaries* deve quindi sempre includere:

- una stringa (*BCName*) che identifica il tipo di condizione al contorno;
- una stringa (*BCIdentifier*) che identifica il codice della condizione al contorno che viene memorizzato all'interno dell'oggetto *db*;
- il numero di entità vincolate per ciascun grado di libertà;
- l'indice dell'entità a cui è assegnata la condizione al contorno e il suo valore, per ciascun grado di libertà;
- la stringa *End* che indica la fine del set di dati.

```
1  NodeBC                                   % BC name
2  nodeDir                                  % BC type
3  3    2     0                             % #ID_blocked, #ID_blocked ,#ID_blocked
4  1    0                                   % ID_x, value_x
5  4    0
6  5    0
7  4    0                                   % ID_y, value_y
8  6    0
9                                           % ID_z, value_z
10 End
```

Figura 4.4: Esempio di file input per la classe *Boundaries*.

Classe *NodeBC*

La classe *NodeBC* gestisce le condizioni al contorno assegnate direttamente sui nodi della mesh. All'interno delle proprietà (di tipo *public*) vengono inizializzati:

- **numID**: un vettore le cui componenti rappresentano il numero totale di nodi vincolati per ciascun grado di libertà;
- **ID_x**, **ID_y**, **ID_z**: vettori che contengono l'indice dei nodi vincolati, ognuno dei tre riferito ad un grado di libertà;
- **dofmax**: numero di gradi di libertà per ciascun nodo, pari a 3;
- **value_x**, **value_y**, **value_z**: vettori che contengono il valore della funzione incognita, ognuno dei tre riferito ad un grado di libertà;
- **bound_dof**: vettore degli indici dei gradi di libertà vincolati;
- **bound_value**: vettore dei valori della funzione incognita, per ciascun grado di libertà vincolato;

La classe *NodeBC* conta metodi di tipo *public* e metodi di tipo *private*:

- la function *NodeBC* è il costruttore della classe che inizializza l'oggetto e rimanda alla function *setBCInput* per l'assegnazione delle proprietà dell'oggetto, a partire dai dati del file di input (figura 4.4);
- la function *NodeBoundary* genera due vettori necessari all'imposizione delle condizioni al contorno, **bound_dof** e **bound_value**.

4.2.4 Elementi finiti

Sono state implementate due classi per la gestione dei parametri degli elementi finiti tridimensionali, attualmente tetraedri lineari ed esaedri trilineari.

Classe *Elements*

La classe *Elements* è la classe che si occupa dell'individuazione del tipo di elemento finito, a partire dal codice specificato nel file di input della mesh (figura 4.2). Le proprietà contano due parametri:

- **elemType**: codice che identifica il tipo di elemento finito;
- **data**: cella che contiene i dati dell'elemento, ovvero coordinate nodali e topologia degli elementi finiti;

La classe *Elements* conta metodi di tipo *public* e metodi di tipo *private*:

- la function *Elements* è il costruttore della classe che inizializza l'oggetto e richiama la function *setElementData* per l'assegnazione dei dati degli elementi;
- la function *getElement* utilizza *elemType* per chiamare la classe che gestisce gli elementi finiti di cui è costituita la mesh.

Non c'è bisogno di nessun file di input dato che tutte le informazioni necessarie alla creazione degli oggetti con le classi *Materials*, *Tetrahedron* e *Hexahedron* si ottengono a partire dalle proprietà dell'oggetto creato con la classe *Mesh*.

Classe *Tetrahedron*

La classe *Tetrahedron* gestisce gli elementi finiti tetraedrici, al momento sono implementati solo i tetraedri lineari a 4 nodi. All'interno delle proprietà (di tipo *public*) vengono inizializzati:

- **dofmax**: massimo numero di gradi di libertà per nodo, pari a 3;
- **nFace**: numero di facce di ciascun elemento, pari a 4;
- **nNode**: numero di nodi dell'elemento;

- **nodeCoords**: coordinate dei nodi della mesh;
- **nElem**: numero totale di elementi che compongono la mesh;
- **elemTopol**: sequenze di nodi che identificano ciascun elemento;
- **surfTopol**: sequenze di nodi che identificano le facce degli elementi;
- **Vol**: volume di ciascun elemento;
- **Area**: area delle facce degli elementi;
- **N**: matrice delle funzioni base;
- **B**: matrice delle derivate delle funzioni base.

Le prime proprietà sono ottenute a partire dai dati della mesh, mentre *Vol, Area, N, B* sono calcolate all'interno della classe, qualora la risoluzione del problema le richieda.
La classe conta un metodo di tipo *private* per l'assegnazione delle proprietà dell'oggetto. All'interno dei metodi di tipo *public*, invece, è implementato il costruttore della classe (function *Tetrahedron*) e altre function:

- *getBasisF* per il calcolo delle funzioni di base dell'elemento;
- *getVolume* per il calcolo del volume degli elementi della mesh;
- *getDerivatives* per il calcolo delle derivate delle funzioni di base;
- *getArea* per il calcolo dell'area delle facce degli elementi.

Classe *Hexahedron*

La classe *Hexahedron* gestisce gli elementi finiti esaedrici, al momento sono implementati solo gli esaedri trilineari a 8 nodi. All'interno delle proprietà (di tipo *public*) vengono inizializzati:

- **dofmax**: massimo numero di gradi di libertà per nodo, pari a 3;
- **dim**: dimensione degli elementi finiti, pari a 3;
- **nFace**: numero di facce di ciascun elemento, pari a 6;
- **nNode**: numero di nodi dell'elemento;
- **nodeCoords**: coordinate dei nodi della mesh;
- **nElem**: numero totale di elementi che compongono la mesh;
- **elemTopol**: sequenze di nodi che identificano ciascun elemento;
- **surfTopol**: sequenze di nodi che identificano le facce degli elementi;

- **cellCentroid**: coordinate del baricentro di ciascun elemento;
- **S_lenght**: misura dei lati degli elementi finiti;
- **Vol**: volume di ciascun elemento;
- **Area**: area delle facce degli elementi;
- **N**: matrice delle funzioni base;
- **B**: matrice delle derivate delle funzioni base;
- **J**: matrice jacobiana degli elementi;
- **derMatrix**: matrice delle derivate delle funzioni base rispetto alle coordinate del sistema naturale.

Le prime proprietà sono ottenute a partire dai dati della mesh, mentre *Vol, Area, N, B, J, derMatrix* sono calcolate all'interno della classe, qualora la risoluzione del problema le richieda. La classe conta un metodo di tipo *private* per l'assegnazione delle proprietà dell'oggetto. All'interno dei metodi di tipo *public*, invece, è implementato il costruttore della classe (function *Hexahedron*) e altre function, già introdotte nella spiegazione della classe *Tetrahedron*, per il calcolo delle funzioni di base dell'elemento, del volume degli elementi della mesh, delle derivate delle funzioni di base e per il calcolo dell'area delle facce degli elementi. In aggiunta a queste, si sono implementate:

- *getSide* per il calcolo della misura dei lati dell'elemento;
- *reOrderTopol* che riordina la sequenza dei nodi dell'elemento in modo che sia congruente con la numerazione dei nodi dell'esaedro di riferimento;
- *getJacobian* per il calcolo della jacobiana.

4.2.5 Function di assemblaggio e imposizione delle condizioni al contorno

Sono state implementate due function per l'assemblaggio della matrice di rigidezza globale e una che gestisce l'imposizione delle condizioni al contorno. La prima function di assemblaggio implementata è *assemblyK* ed è utilizzabile solo per i tetraedri lineari. Gli oggetti di input richiesti dalla funzione, da fornire nell'ordine sotto indicato, sono:

- il numero totale di nodi della mesh (che coincide con la proprietà **nNodes** della classe *Mesh*);
- l'indice di materiale di ciascun elemento (che coincide con la proprietà **cellTag** della classe *Mesh*);
- l'oggetto generato con la classe **Materials** per l'identificazione del materiale e, di conseguenza, il calcolo della matrice costituitiva corretta;

- l'oggetto creato con la classe **Tetrahedron** per il numero di gradi di libertà di ciascun nodo, il numero totale di elementi nella mesh, la topologia di ciascun elemento, il volume degli elementi e la matrice delle derivate delle funzioni di base.

La function accetta anche altri dati di input, oltre a quelli appena elencati che sono strettamente necessari, nel caso in cui il legame costitutivo del materiale li richieda. La function calcola la matrice di rigidezza locale di ciascun elemento e ne assegna i contributi nella posizione corretta della matrice locale.

Le seconda function di assemblaggio implementata è *assemblyKGauss* e differisce dalla prima in termini di assemblaggio della matrice di rigidezza locale dell'elemento. All'interno di questa seconda function, infatti, si effettua l'integrazione della matrice mediante la tecnica della quadratura di Gauss, riportata al paragrafo 2.4.2. L'assemblaggio della matrice di rigidezza globale e gli oggetti di input rimangono invariati, fatta eccezione per l'oggetto **Tetrahedron** che viene sostituito da un oggetto creato con la classe **Hexahedron**.

La function per l'imposizione delle condizioni al contorno *imposeBC* vincola determinati gradi di libertà all'interno della matrice di rigidezza globale, assegnando loro un valore, e assembla il vettore dei termini noti. I dati di input richiesti dalla funzione, da fornire nell'ordine sotto indicato, sono:

- il numero totale di nodi della mesh;
- la matrice di rigidezza globale;
- l'oggetto che contiene le condizioni al contorno di Dirichlet per l'indice dei gradi di libertà vincolati e il valore ad essi assegnato;
- l'oggetto che contiene le condizioni al contorno di Neumann per l'indice dei gradi di libertà sui quali agisce la forzante e il valore della forzante ad essi assegnato.

Una volta imposte anche le condizioni al contorno, si calcola il vettore degli spostamenti $\mathbf{s}$ risolvendo il sistema lineare:

$$H\mathbf{s} = \mathbf{r} \tag{4.1}$$

Per l'avvio di una simulazione in GReS sono necessari i seguenti file di *input*:

1. File contenente le dimensioni e le caratteristiche della mesh (es: *mesh.msh*);
2. File contenente i parametri meccanici e il tipo di legame costituitivo del terreno (es: *materials.dat*);
3. File contentente le condizioni di vincolo sugli spostamenti (es: *dirNode.dat*);
4. File contenente le forzanti esterne (es: *neuNode.dat*).

Capitolo 5

Test di validazione

Al termine dell'implementazione delle classi riportate nel capitolo precedente, si sono eseguiti una serie di test finalizzati alla validazione del codice. In particolare, l'obbiettivo dei primi test è stato quello di verificare il funzionamento generale, che implica:

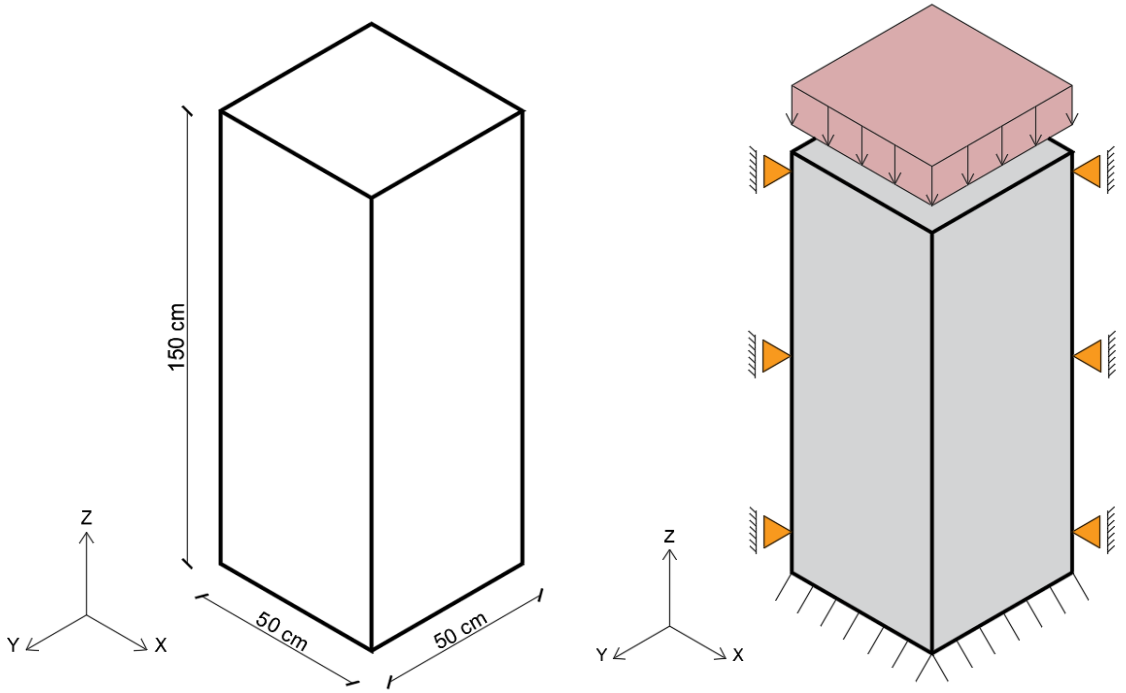
- la lettura dei dati del dominio discretizzato;
- il calcolo della matrice costitutiva dei materiali;
- la corretta implementazione degli elementi tridimensionali in cui è suddivisa la mesh (tetraedri lineari ed esaedri trilineari);
- l'assemblaggio della matrice di rigidezza globale;
- l'imposizione delle condizioni al contorno del problema.

La soluzione ottenuta con GReS è stata confrontata con la soluzione analitica del problema analizzato, in modo da valutare l'entità dell'errore tra i risultati del modello e quelli analitici.

Una volta validato il codice, si sono svolti ulteriori test in cui si è impiegato il modello costitutivo ipoelastico e si è effettuato l'assemblaggio di due materiali. A differenza di quanto avvenuto per i primi casi studio, non c'è stato un confronto diretto con la soluzione analitica.

5.1 Validazione del codice

Per validare il codice si è scelto di simulare lo schiacciamento di un provino di terreno. Il provino è soggetto ad una pressione distribuita uniformemente su tutta l'area superficiale, pari a 200 kPa, e vincolato in modo tale che gli spostamenti fuori dal piano lungo il perimetro e il fondo siano impediti. Il provino è un parallelepipedo a base quadrata, di dimensione $50x50$ cm^2, di altezza pari a 150 cm. Lo schema del modello, comprensivo di dimensioni, vincoli e forzanti, è riportato in figura 5.1. Il materiale del modello è un limo sabbioso, caratterizzato da un modulo di Young pari a 15 MPa e un coefficiente di Poisson pari a 0.3. Si ipotizza un comportamento elastico lineare isotropo per il materiale.



(a) Dimensioni del provino **(b)** Modello del provino utilizzato

Figura 5.1: Provino di terreno di dimensioni 50x50x150cm, vincolato e soggetto ad una pressione superficiale uniforme.

5.1.1 Soluzione analitica

Secondo la teoria del problema elastico, la superficie caricata del provino si abbassa uniformemente di una quantità pari a:

$$\Delta u_z = \varepsilon_z H \tag{5.1}$$

in cui ε_z è la deformazione verticale del provino e H è l'altezza iniziale del provino. Dalla legge costitutiva di un materiale elastico lineare e isotropo riportata in (3.1.1), si ottiene che:

$$\sigma_z = \frac{E(1-\nu)}{(1+\nu)(1-2\nu)}\varepsilon_z \tag{5.2}$$

Il termine che moltiplica la componente di deformazione rappresenta la compressibilità del materiale, e si denota con K. A questo punto lo spostamento verticale del provino risulta

$$\Delta u_z = \frac{\sigma_z}{K} H \tag{5.3}$$

Il materiale di cui è costituito il provino è caratterizzato da una compressibilità $K = 20.19\,MPa$ e l'abbassamento del provino è dunque pari a $\Delta u_z = -14.86\,mm$.

5.1.2 Test 1 - Tetraedri lineari

Per il primo test si sceglie di discretizzare il modello descritto sopra (5.1) con elementi tetraedrici lineari (4 nodi). Si è scelto di realizzare due mesh distinte, che differiscono per grado di raffinamento: *tetra1* e *tetra2*. In questo modo è stato possibile valutare il grado di approssimazione della soluzione all'aumentare del numero di nodi.

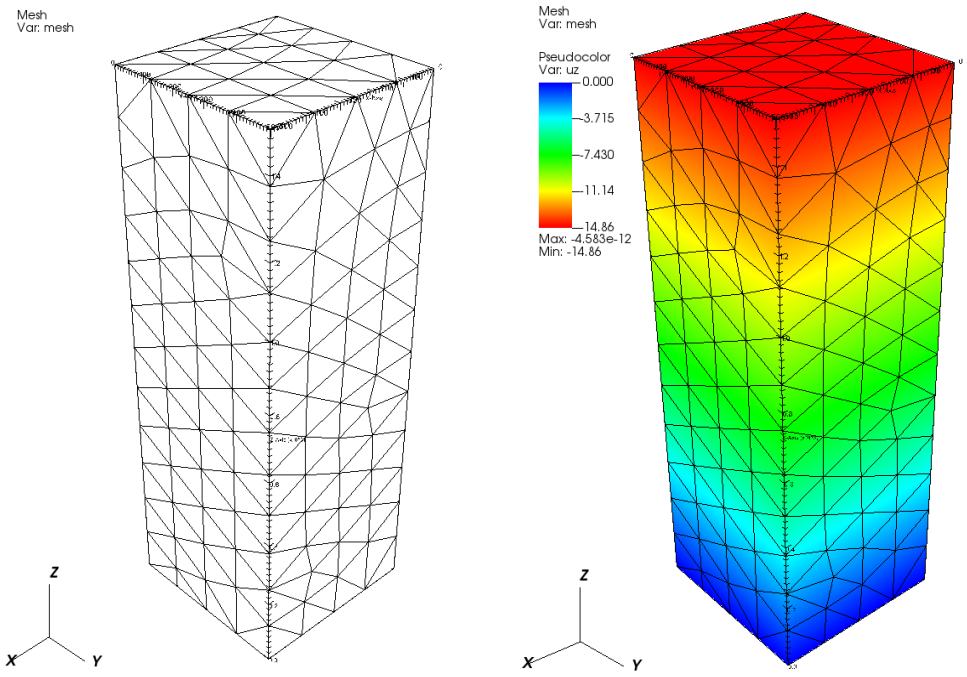


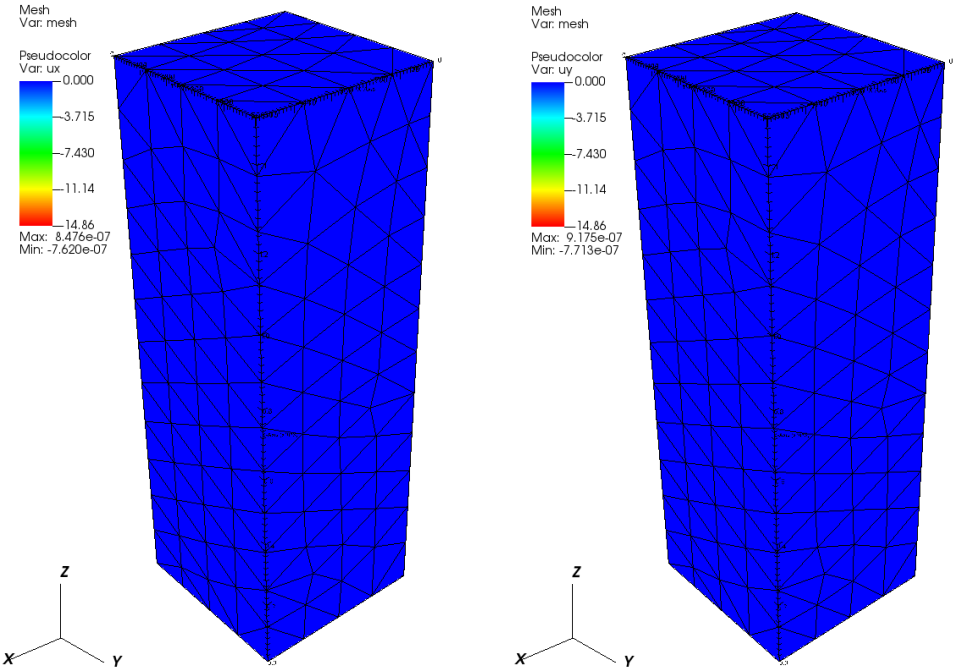
Figura 5.2: Geometria e andamento degli spostamenti lungo l'asse z della mesh *tetra1*.

La mesh *tetra1* (figura 5.2) conta un totale di *323 nodi* e *1121 elementi*, per un totale di *969 incognite* (ad ogni nodo sono infatti associati 3 gradi di libertà).

Una volta ripartito il valore della pressione agente sui nodi superficiali, si è ottenuto uno spostamento uniforme su questi ultimi, pari a $\Delta u_z = -14.86\ mm$. Il valore ottenuto coincide esattamente con la soluzione analitica. Per quanto riguarda invece gli spostamenti orizzontali, questi sono vincolati e risultano in effetti nulli. L'andamento dei valori di spostamento ottenuti con GReS nelle tre direzioni è riportato nelle figure 5.2 e 5.3. È evidente che lo spostamento sommitale del provino sia pari alla soluzione analitica, e che il suo valore diminuisca linearmente man mano che ci si avvicina alla base vincolata. Gli spostamenti nel piano risultano nulli. Per una misura quantitativa dell'errore tra la soluzione analitica e la soluzione approssimata, si valuta il valore dell'integrale:

$$e = \int_{\Omega} (\hat{\mathbf{u}} - \bar{\mathbf{u}})\, d\Omega \tag{5.4}$$

in cui $\bar{\mathbf{u}}$ rappresenta il vettore della soluzione analitica e $\hat{\mathbf{u}}$ è il vettore della soluzione appros-



(a) Spostamenti lungo l'asse x **(b)** Spostamenti lungo l'asse y

Figura 5.3: Andamento degli spostamenti nel piano (x,y) per la mesh *tetra1*.

simata. Generalmente la soluzione approssimata si calcola come:

$$\hat{\mathbf{u}} = N\mathbf{s} \tag{5.5}$$

in cui $\mathbf{s}$ è il vettore degli spostamenti nodali e N è la matrice delle funzioni di base. Per gli elementi finiti finora implementati, la soluzione esatta è lineare, quindi la misura dell'errore si può valutare più semplicemente calcolando la norma euclidea della differenza tra la soluzione approssimata e quella esatta:

$$e = \frac{\| \hat{\mathbf{u}} - \bar{\mathbf{u}} \|_2}{\| \bar{\mathbf{u}} \|_2} = 1.78 \cdot 10^{-7} \tag{5.6}$$

Si può concludere, dunque, che la soluzione calcolata con GReS coincide con la soluzione analitica in tutto il dominio. Il numero che risulta dalla (5.6) è condizionato dalla precisione di macchina.

Sebbene lo spostamento del modello coincida già con la soluzione analitica, si è realizzata una mesh di tetraedri più raffinata, *tetra2* (figura 5.4). Questa mesh conta un totale di *1601 nodi* e *6485 elementi*, per un totale di *4803 incognite*. Si è ricalcolato il valore della pressione agente su ciascun nodo superficiale e si è ottenuto uno spostamento sommitale uniforme pari a $\Delta u_z = -14.86\ mm$. Il valore ottenuto coincide nuovamente con la soluzione analitica mentre gli spostamenti orizzontali risultano nulli. L'andamento dei valori di spostamento ottenuti con GReS nelle tre direzioni è riportato nelle figure 5.4 e 5.5.

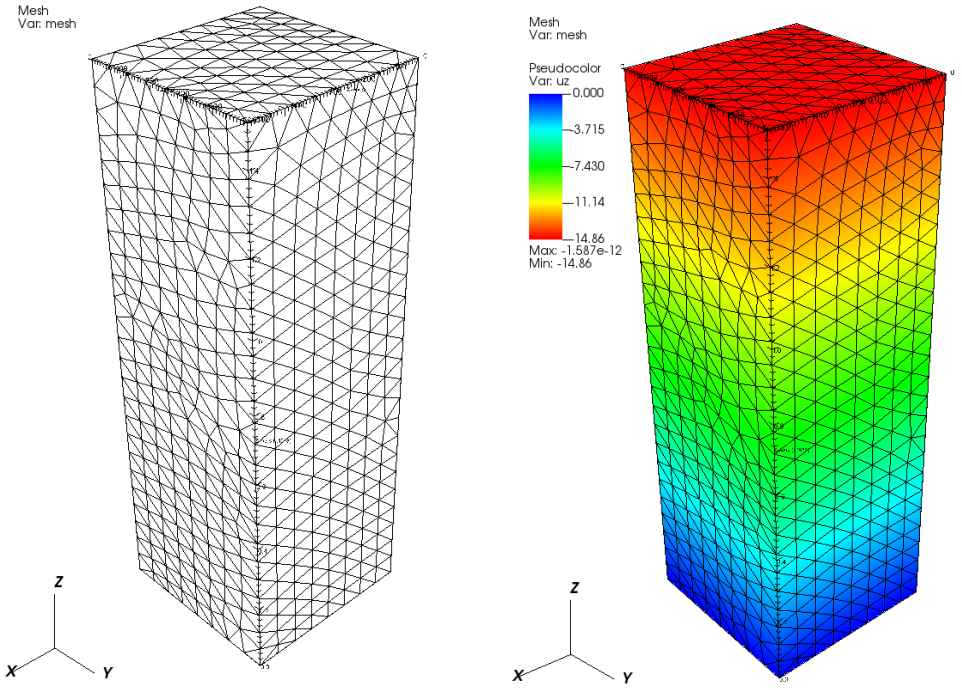
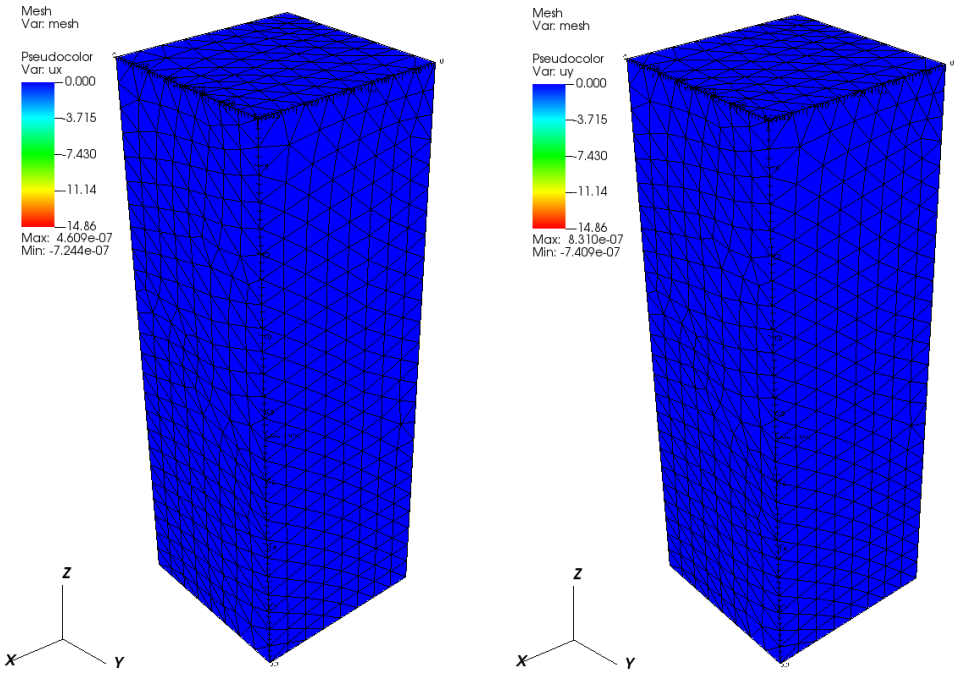


Figura 5.4: Geometria e andamento degli spostamenti lungo l'asse z della mesh *tetra2*.



(a) Spostamenti lungo l'asse x **(b)** Spostamenti lungo l'asse y

Figura 5.5: Andamento degli spostamenti nel piano (x,y) per la mesh *tetra2*.

L'errore tra la soluzione analitica e quella approssimata calcolato come in (5.6) è pari a: $8.03 \cdot 10^{-8}$.

5.1.3 Test 2 - Esaedri trilineari

Per il secondo test si sceglie, invece, di discretizzare il modello descritto al paragrafo 5.1 con elementi esaedrici trilineari (8 nodi). Anche in questo caso, si è scelto di realizzare due mesh che differiscono per grado di raffinamento (*hexa1* e *hexa2*) in modo da valutare il grado di approssimazione della soluzione all'aumentare del numero di nodi.

La mesh *hexa1* (figura 5.6) conta un totale di *325 nodi* e *192 elementi*, per un totale di *975 incognite* (ad ogni nodo sono infatti associati 3 gradi di libertà). Una volta ripartito il valore della pressione agente sui nodi superficiali, si è ottenuto uno spostamento uniforme su questi ultimi, pari a $\Delta u_z = -14.86\ mm$. Il valore ottenuto coincide esattamente con la soluzione

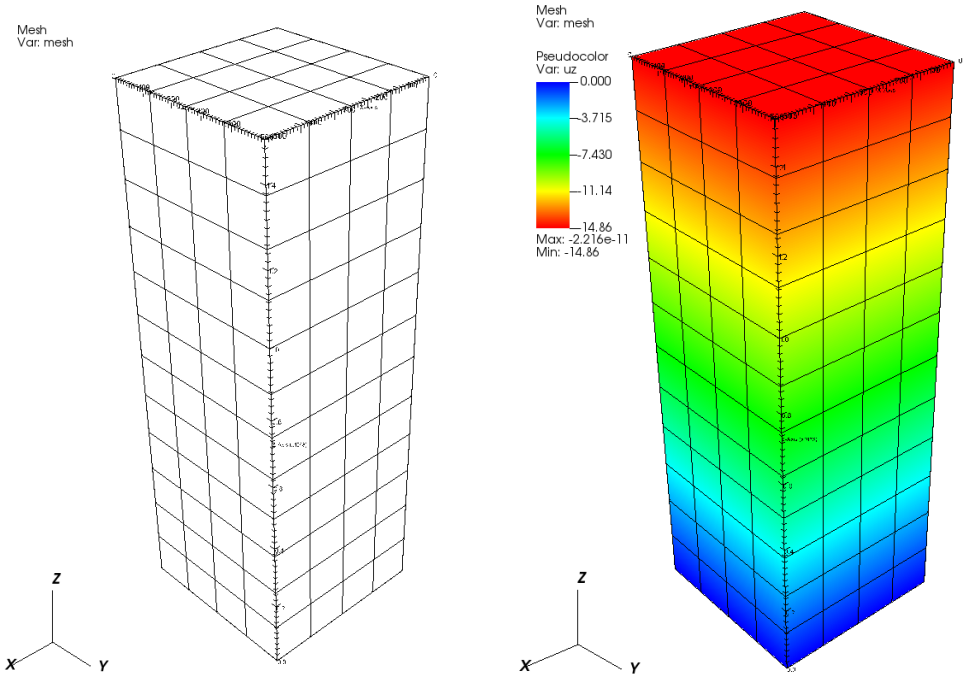
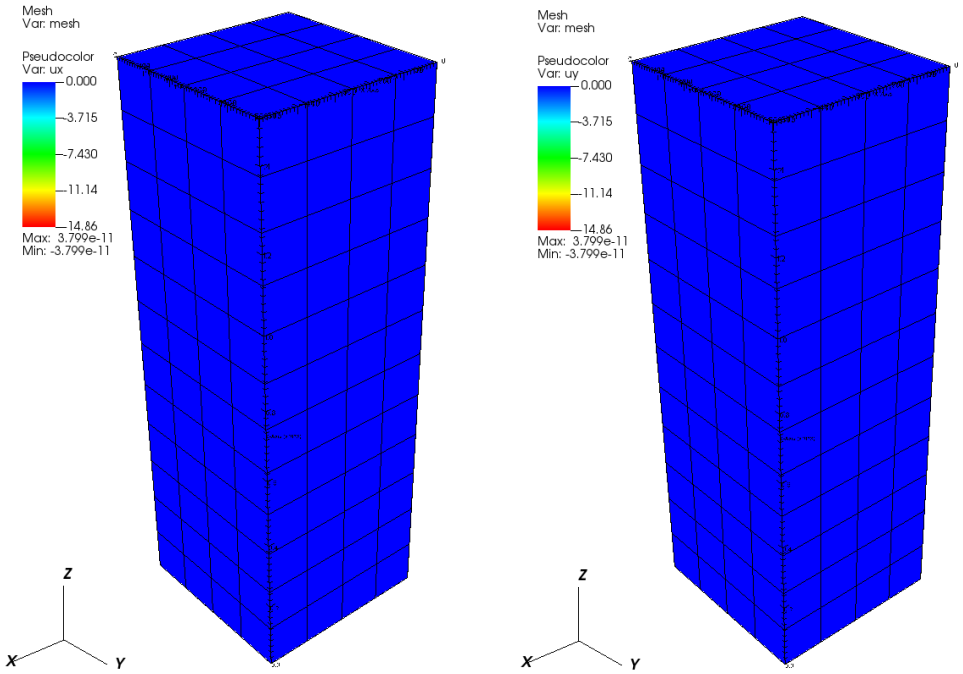


Figura 5.6: Geometria e andamento degli spostamenti lungo l'asse z della mesh *hexa1*.

analitica e con la soluzione calcolata per le mesh discretizzate con i tetraedri. Per quanto riguarda invece gli spostamenti orizzontali, questi sono vincolati e risultano in effetti nulli. L'andamento dei valori di spostamento ottenuti con GReS nelle tre direzioni è riportato in figura 5.6 e 5.7. Come avvenuto per i tetraedri, lo spostamento sommitale del provino risulta pari alla soluzione analitica, e il suo valore va diminuendo man mano che ci si avvicina alla base vincolata. Gli spostamenti nel piano risultano nulli.



(a) Spostamenti lungo l'asse x **(b)** Spostamenti lungo l'asse y

Figura 5.7: Andamento degli spostamenti nel piano (x,y) per la mesh *hexa1*.

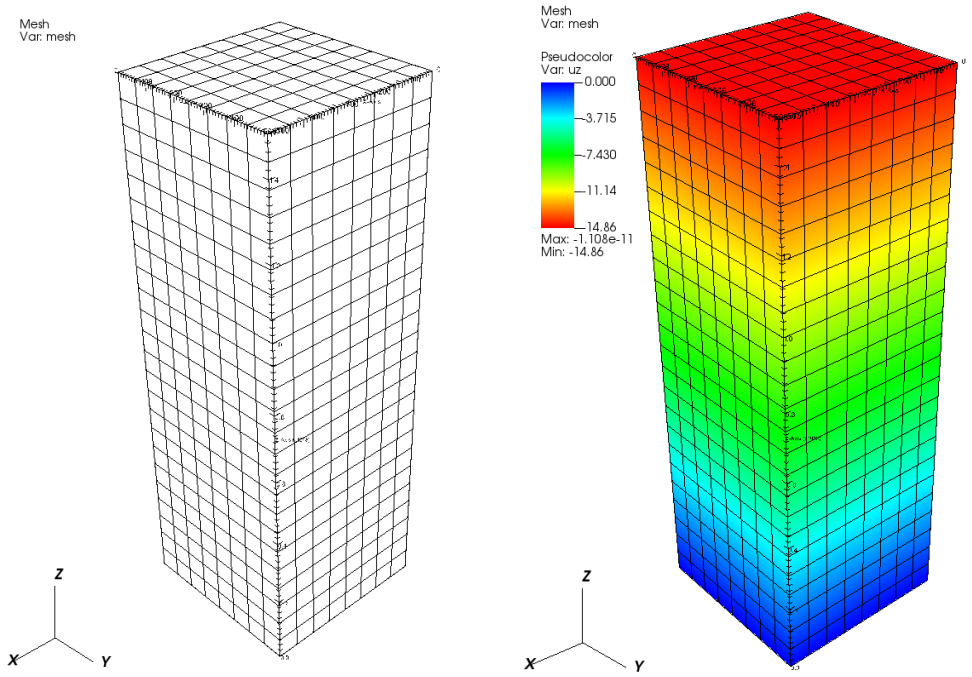
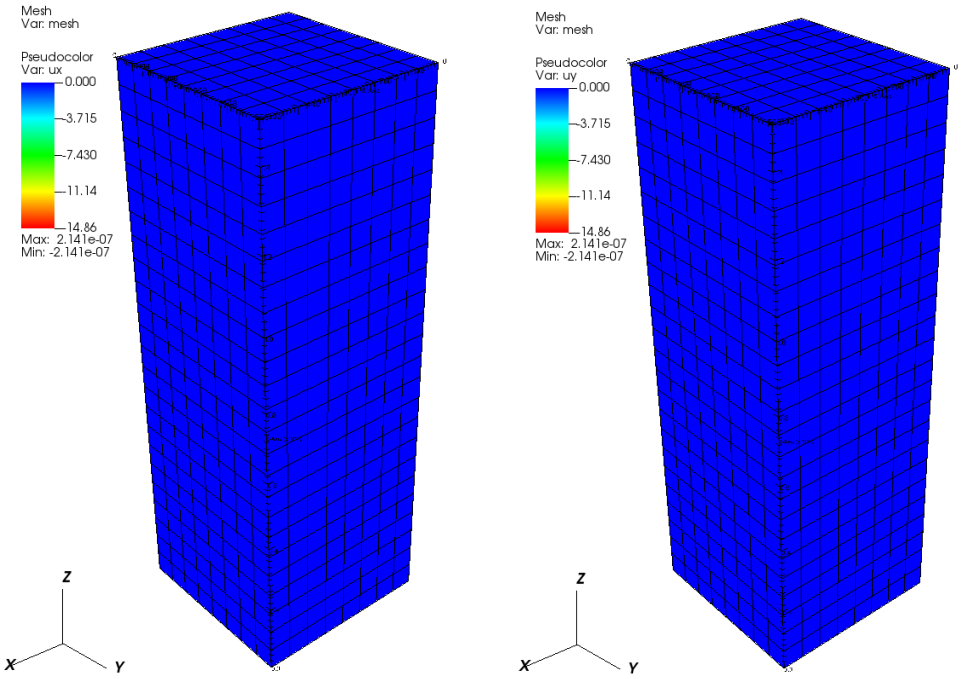


Figura 5.8: Geometria della mesh *hexa2*.

L'errore tra la soluzione analitica e quella approssimata calcolato come in (5.6) è pari a: $1.8 \cdot 10^{-11}$. Sebbene lo spostamento del modello coincida già con la soluzione analitica, si è realizzata una mesh di esaedri più raffinata, *hexa2* (5.8). Questa mesh conta un totale di *2025 nodi* e *1536 elementi*, per un totale di *6075 incognite*.

Si è ricalcolato il valore della pressione agente su ciascun nodo superficiale e si è ottenuto uno spostamento sommitale uniforme pari a $\Delta u_z = -14.86\ mm$. Il valore ottenuto coincide nuovamente con la soluzione analitica mentre gli spostamenti orizzontali risultano nulli. L'andamento dei valori di spostamento ottenuti con GReS nelle tre direzioni è riportato nelle figure 5.8 e 5.9.



(a) Spostamenti lungo l'asse x **(b)** Spostamenti lungo l'asse y

Figura 5.9: Andamento degli spostamenti nel piano (x,y) per la mesh *hexa2*.

L'errore tra la soluzione analitica e quella approssimata calcolato come in (5.6) è pari a: $4.5 \cdot 10^{-8}$.

5.1.4 Test 3 - Modello costitutivo elastico lineare trasversalmente isotropo

Terminati i test di validazione utilizzando gli elementi tetraedrici ed esaedrici con un materiale elastico lineare ed isotropo, si è effettuato un ultimo test di validazione utilizzando la mesh tetraedrica più raffinata, *tetra2*, e il legame costitutivo *elastico lineare trasversalmente isotropo*. I vincoli sono gli stessi utilizzati per i precedenti test di validazione, il carico agente in superficie è pari a 100 *kPa* e si assumono:

- modulo di Poisson omogeneo $\nu_H = \nu_Z = 0.3$;
- modulo di Young lungo la direzione verticale z pari a 15 *MPa*;
- modulo di Young nel piano (x,y) pari a 1 *MPa*.

La soluzione analitica è quella descritta nel paragrafo 5.1.1, tenendo conto che in questo caso la compressibilità del mezzo K, secondo quanto riportato al paragrafo 3.1.2, equivale a 5.25 *MPa*. L'abbassamento del provino è quindi pari a $\Delta u_z = -28.57\ mm$.

Il valore di spostamento in sommità del provino ottenuto in GReS coincide con la soluzione analitica ed è pari a $\Delta u_z = -28.57\ mm$, come mostrato in figura 5.10, mentre gli spostamenti orizzontali risultano nulli, come nei test precedenti.

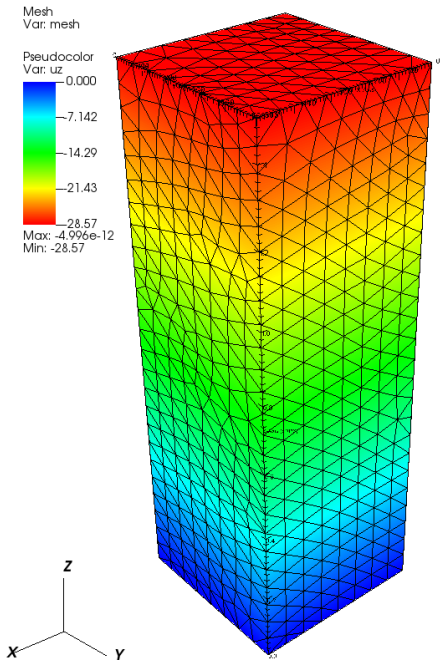


Figura 5.10: Materiale elastico lineare trasversalmente isotropo: andamento degli spostamenti lungo l'asse z per la mesh *tetra2*.

L'errore tra la soluzione analitica e quella approssimata calcolato come in (5.6) è pari a: $2.6 \cdot 10^{-6}$.

Dopo aver validato GReS, si sono svolti altri due test con lo scopo di valutare l'implementazione del modello costitutivo ipoelastico e l'assemblaggio di più materiali all'interno di uno stesso dominio. La geometria e i vincoli del modello sono quelli descritti al paragrafo 5.1. Per quanto riguarda la mesh, si è impiegata la mesh discretizzata con elementi esaedrici trilineari, *hexa3*, mentre il valore del carico agente in superficie e i parametri meccanici del materiale variano a seconda del test.

5.1.5 Test 4 - Modello costitutivo ipoelastico

Il legame costitutivo ipoelastico è un legame elastico *non lineare* in cui la compressibilità del mezzo dipende dal valore della tensione verticale σ_z. Non potendo fare un confronto con una soluzione analitica, come avvenuto per il legame costitutivo elastico lineare, si sono confrontati i risultati ottenuti da diversi test, al variare del valore della tensione di precompressione del mezzo. Data la non linearità del legame costitutivo, sarebbe necessario utilizzare un solutore non lineare per il calcolo degli spostamenti. Non essendo questo ancora implementato in GReS, si sono calcolati gli spostamenti per una configurazione iniziale in cui l'andamento delle tensioni è noto. Il provino è soggetto ad un carico superficiale uniformemente distri-

buito $\sigma_{sup} = 50\ MPa$, e si è assunto un andamento delle tensioni verticali lineare, dovuto al peso proprio del materiale del provino:

$$\sigma_w(z) = \gamma \cdot z \tag{5.7}$$

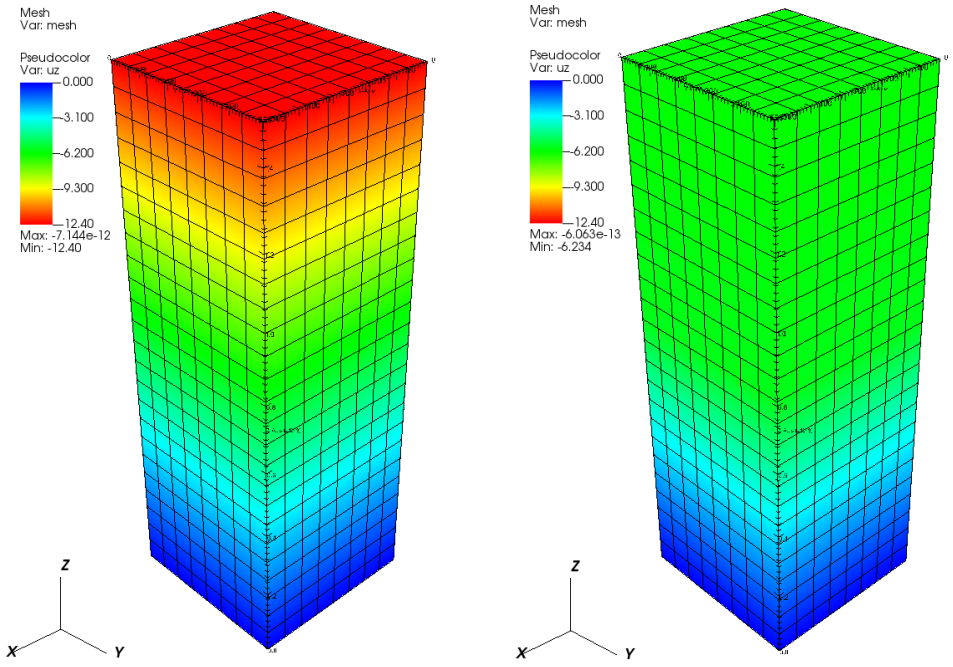
in cui γ è il peso specifico del terreno, pari a 21.5 kN/m^3. Il legame ipoelastico necessita, inoltre, di un modulo di Poisson, pari a 0.3 e di altri 4 parametri per il calcolo della compressibilità verticale, come spiegato al paragrafo 3.1.3. A partire dai valori riportati in [Spiezia et al., 2017], si sono assunti:

$$a = 2 \cdot 10^{-2} \qquad b = -1.15$$
$$a' = 3 \cdot 10^{-4} \qquad b' = -0.8$$

Si sono eseguiti tre test andando a variare il valore della tensione di precompressione σ_P, in particolare:

- *Primo test*: provino completamente in compressione vergine ($\sigma_P = 40\ MPa,\ \sigma_P < \sigma_{sup}$);
- *Secondo test*: provino per metà in compressione vergine e per metà in fase di ricompressione ($\sigma_P = 70\ MPa,\ \sigma_{sup} < \sigma_P < \sigma_{sup} + \sigma_w(150)$);
- *Terzo test*: provino completamente in fase di ricompressione ($\sigma_P = 85\ MPa,\ \sigma_P > \sigma_{sup} + \sigma_w(150)$).

I risultati ottenuti sono riportati nelle figure 5.11. Nel primo test il provino sviluppa un



(a) Compressione vergine

(b) Metà fase vergine e fase metà di ricompressione

Figura 5.11: Materiale ipoelastico: andamento degli spostamenti lungo l'asse z.

abbassamento sommitale pari a $\Delta u_z = -12.40\ mm$. Se il legame ipoelastico è implementato correttamente, questo valore di spostamento deve essere maggiore degli spostamenti che si svilupperanno negli altri test, dato che il valore di compressibilità in un provino in fase vergine è maggiore di quello in un provino in parziale ricompressione. Gli abbassamenti, inoltre, hanno un andamento lineare, poiché una volta fissati il valore del carico superficiale e delle costanti a e b, la compressibilità dipende esclusivamente dall'andamento della tensione verticale, anch'esso lineare.

Nel secondo test, si nota come il valore degli abbassamenti in sommità risulti minore rispetto al primo test, $\Delta u_z = -6.23\ mm$. Questo è dovuto al fatto che il terreno nella metà inferiore del provino è in fase di ricompressione, di conseguenza la compressibilità in quella zona è diminuita rispetto al test precedente e il terreno nella metà inferiore è più rigido. Gli abbassamenti si concentrano nella metà superiore del provino in fase di compressione vergine, in cui la compressibilità è maggiore.

Infine, nel terzo test, il provino è completamente in fase di ricompressione. L'abbassamento sommitale è inferiore rispetto a quello calcolato nei test precedenti, $\Delta u_z = -0.80\ mm$, perché il provino è meno comprimibile. In conclusione, i risultati ottenuti per il legame ipoelastico sono ammissibili e si ritiene dunque che l'implementazione sia corretta.

5.1.6 Test 5 - Assemblaggio di diversi materiali

Infine, si è testato l'assemblaggio di due materiali elastici lineari isotropi, all'interno della stessa mesh. A partire dalla mesh *hexa3* utilizzata per il test al paragrafo 5.1.3, caratterizzata da un modulo di Young pari a 15 *MPa*, si è inserito uno strato centrale di materiale più rigido. In particolare, si è assegnata una rigidezza pari a 50 *MPa* e poi una rigidezza pari a 100 *MPa*, mentre il modulo di Young dello strato soprastante e sottostante si è mantenuto invariato e pari a 15 *MPa*. Il modulo di Poisson è sempre pari a 0.3. Si sono poi valutati gli abbassamenti del provino lungo tutta la sua altezza, in corrispondenza del nodo centrale della mesh.

Il primo grafico (figura 5.13) mostra il confronto tra l'andamento degli abbassamenti per i due provini in figura 5.12. Il primo provino è omogeneo e caratterizzato da $E_1 = E_2 = E = 15\ MPa$, mentre il secondo provino è eterogeneo e caratterizzato da uno strato centrale più rigido in cui $E_2 = 50\ MPa$. Gli strati sovrastante e sottostante del provino eterogeneo sono caratterizzati da $E_1 = 15\ MPa$.

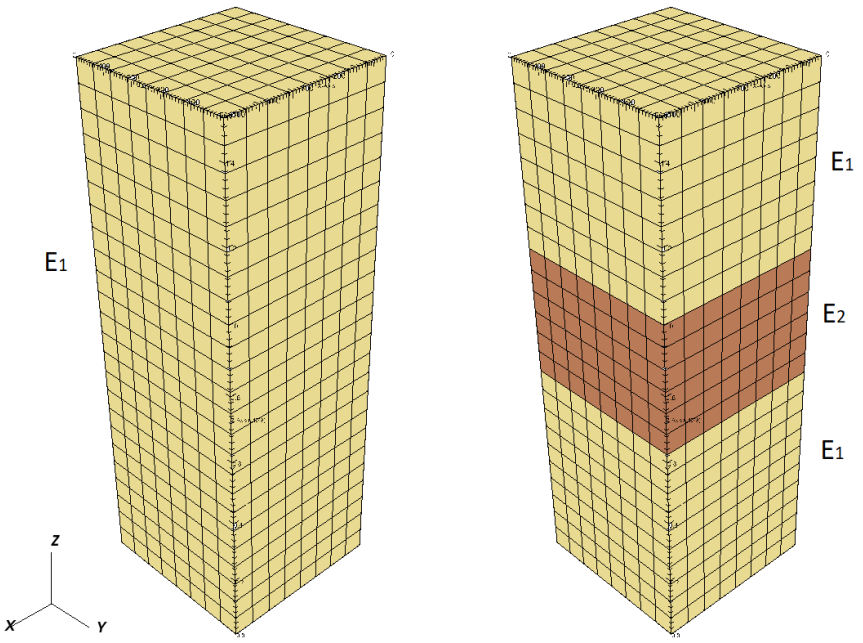


Figura 5.12: Provino di terreno omogeneo caratterizzato da un materiale elastico lineare isotropo con $E_1 = 15\ MPa$ e provino di terreno eterogeneo con uno strato intermedio più rigido in cui $E_2 > 15\ MPa$.

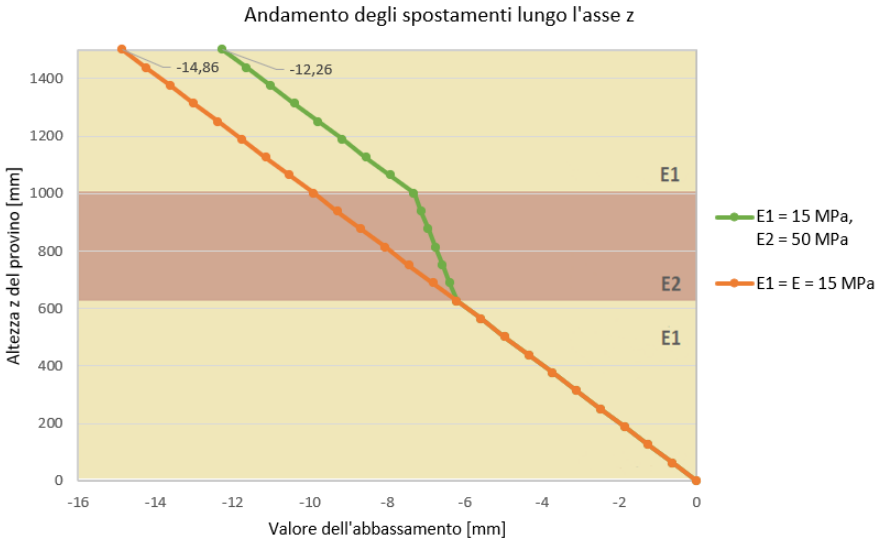


Figura 5.13: Confronto tra l'andamento degli spostamenti di un provino omogeneo con $E_1 = 15\ MPa$, e un provino eterogeneo con uno strato intermedio più rigido caratterizzato da $E_2 = 50\ MPa$.

Risulta evidente che la presenza di uno strato di materiale più rigido condiziona la compressibilità del provino, che sviluppa un abbassamento sommitale di $\Delta u_z = -12.26\ mm$, inferiore rispetto a quello del provino omogeneo ($\Delta u_z = -14.86\ mm$). L'andamento degli abbassamenti si mantiene lineare però, in corrispondenza della variazione di rigidezza dello strato centrale, si nota una variazione di pendenza della retta, dovuta al fatto che gli spostamenti

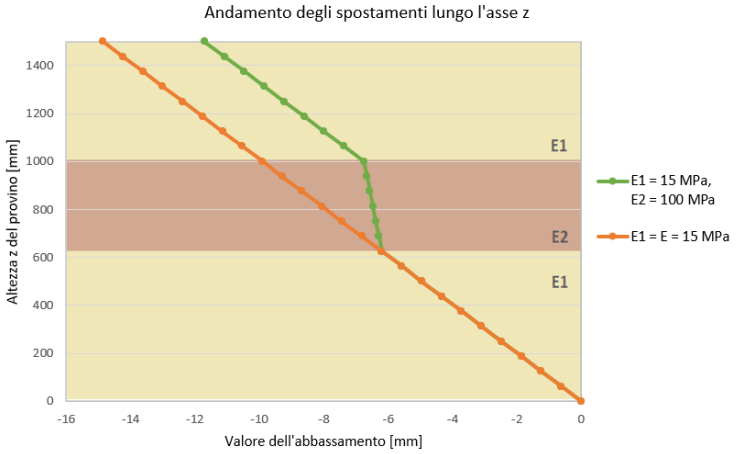


Figura 5.14: Confronto tra l'andamento degli spostamenti di un provino omogeneo con $E_1 = 15\ MPa$, e un provino eterogeneo con uno strato intermedio più rigido caratterizzato da $E_2 = 100\ MPa$.

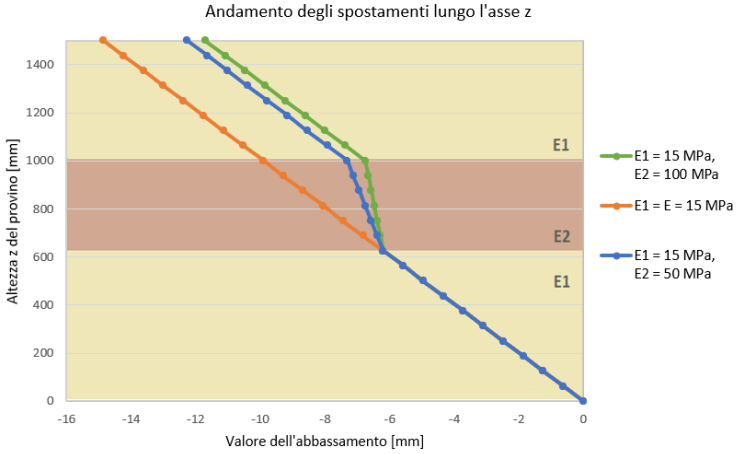


Figura 5.15: Confronto tra l'andamento degli spostamenti di un provino omogeneo caratterizzato da $E = 15\ MPa$, e due provini eterogenei con uno strato intermedio più rigido caratterizzati da $E_2 = 50\ MPa$ e $E_2 = 100\ MPa$.

sono inversamente proporzionali alla compressibilità del mezzo.

Nel secondo grafico (figura 5.14) lo strato più rigido del provino eterogeneo è caratterizzato da un modulo $E_2 = 100\ MPa$. Valgono le stesse considerazioni del caso precedente, l'unica differenza è data dal valore di spostamento sommitale che risulta pari a $\Delta u_z = -11.70\ mm$. La riduzione di abbassamento è giustificata dall'aumento della rigidezza del provino. L'andamento degli spostamenti di tutti i provini è riportato in figura 5.15.

Capitolo 6

Conclusioni

Il presente lavoro di tesi si pone come obbiettivo lo sviluppo preliminare di GReS, un codice generale per la simulazione di problemi di giacimento. Con problemi di giacimento si intendono tutti i processi legati all'iniezione e/o all'estrazione di fluidi nel sottosuolo, che possono innescare fenomeni quali la subsidenza del suolo o la riattivazione delle faglie. Lo scopo di GReS è quello di realizzare uno strumento che tenga conto della molteplicità dei fenomeni fisici coinvolti in un problema di giacimento, alcuni dei quali sono ancora oggi oggetto di ricerca e sviluppo. In questa tesi, che è un primo passo verso la realizzazione di GReS, si è costruita la struttura comune del codice, con la quale è risolto il problema dell'equilibrio elastico. Il modello matematico si basa sul principio dei lavori virtuali, ovvero sul fatto che in presenza di una configurazione deformata virtuale soddisfacente le condizioni di congruenza, l'uguaglianza tra lavoro interno e lavoro esterno è condizione necessaria e sufficiente per l'equilibrio del sistema. Il modello numerico che risolve il problema differenziale associato al principio dei lavori virtuali è il metodo degli elementi finiti, un metodo variazionale che approssima la soluzione del sistema di equazioni differenziali, calcolandone il valore in determinati punti del dominio discreto, detti nodi. Nel caso del problema dell'equilibrio elastico, la soluzione approssimata è data dalle componenti di spostamento u, v, w che minimizzano il funzionale associato all'energia potenziale del sistema 2.10. Pertanto, GReS attualmente calcola lo spostamento dei nodi del dominio discreto mediante la risoluzione del sistema lineare formulato a partire dal principio dei lavori virtuali:

$$H\mathbf{s} = \mathbf{r}$$

in cui H è la matrice di rigidezza del sistema, $\mathbf{s}$ è il vettore degli spostamenti nodali e $\mathbf{r}$ contiene le forze di massa e le forze nodali agenti sul sistema. Il calcolo di una soluzione numerica con il metodo degli elementi finiti prevede diverse fasi. Innanzitutto è necessario discretizzare il dominio, ovvero suddividerlo in elementi finiti, scelti in base alle geometria del dominio e al grado delle funzioni che descrivono il problema fisico oggetto di analisi. In seguito, è necessario definire il legame tra le componenti di tensione e di deformazione del mezzo, ovvero il modello costitutivo, e le condizioni al contorno, ovvero i vincoli e le forzanti agenti

sul sistema. Infine, si assembla la matrice di rigidezza globale, si impongono le condizioni al contorno, e si risolve il sistema lineare. GReS è un codice agli elementi finiti implementato in Matlab, tramite il paradigma di programmazione a classi. Si è scelto di realizzare il codice in Matlab data la sua larga diffusione e il suo basso costo di accesso. Inoltre, permette di richiamare routine più efficienti realizzate in altri linguaggi, garantendo così un'adeguata efficienza computazionale. La programmazione a classi, invece, conferisce modularità al codice. Ogni classe infatti è una struttura dati indipendente, che incapsula determinati dati del problema sui quali opera per mezzo di metodi. Questo aspetto agevola i processi di integrazione del codice anche da parte di più programmatori contemporaneamente. Le classi implementate finora sono indipendenti dalla fisica del problema che si desidera descrivere e gestiscono tutte le fasi del metodo degli elementi finiti riportate sopra. In particolare, GReS è attualmente composto da:

- una classe *Mesh*;
- una classe *Elements* e due classi che gestiscono gli elementi finiti tetraedrici lineari ed esaedrici trilineari *Tetrahedron* ed *Hexahedron*, rispettivamente;
- una classe *Materials* e più classi che si occupano del legame costitutivo assegnato al materiale (*Elastic*, *TransvElastic* e *HypoElastic*) o delle caratteristiche del fluido e del mezzo poroso (*Fluid* e *PorousRock*);
- una classe *Boundaries* e una classe che gestisce le condizioni al contorno nodali *NodeBC*.

Per l'assemblaggio della matrice di rigidezza e l'imposizione delle condizioni al contorno sono state invece implementate tre function, una delle quali opera l'integrazione della matrice di rigidezza sul volume tramite la tecnica di quadratura di Gauss. Una volta ultimata la scrittura del codice, si sono effettuati una serie di test di validazione per valutare il corretto funzionamento di classi e function. Si è confrontata la soluzione calcolata con GReS con la soluzione analitica nota di un problema di compressione edometrica su un provino di terreno. I test di validazione sono stati eseguiti su mesh discretizzate tramite elementi finiti tetraedrici ed esaedrici, sia con un materiale elastico lineare isotropo, sia con un materiale trasversalmente isotropo. L'errore ottenuto tra la soluzione analitica e la soluzione approssimata è stato in tutti i casi dell'ordine di 10^{-6} o minore. Si conclude, quindi, che il codice è stato implementato correttamente, in tutte le sue fasi. Infine, si sono testati anche l'assemblaggio di più materiali e il legame costitutivo ipoplastico. Per questi test non è stato possibile effettuare un confronto diretto con la soluzione analitica, ma i risultati ottenuti sono ammissibili.

Lo scopo del presente lavoro è stato quello di gettare le basi di un codice di simulazione generico per i problemi di giacimento. L'esito positivo dei test di validazione hanno determinato la corretta implementazione di GReS, ma l'obbiettivo del codice è quello di simulare molteplici fisiche legate ai giacimenti. Quindi, a partire da quanto implementato finora, si possono:

- implementare ulteriori modelli costitutivi per il terreno, ad esempio modelli di tipo plastico;
- aggiungere la simulazione di altri fenomeni fisici coinvolti nei problemi di giacimento come ad esempio l'andamento della temperatura o la meccanica del contatto;
- migliorare della performance attuale del codice.

In conclusione, GReS è stato implementato correttamente e a partire da quanto realizzato finora, si possono aggiungere nuove funzionalità, utili alla risoluzione di problemi di giacimento.

Appendice A

Matlab code

Listing A.1: General material class

```
classdef Materials < handle
  % MATERIAL General material class

  properties (Access = public)
    % Creation of a Map object
    db = containers.Map;
  end

  methods (Access = public)
    % Class constructor method
    function obj = Materials(fileName)
      % Calling the function to read input data from file
      obj.readInputFile(fileName)
    end

    % Check if the matIdentifier defined by the user is a key of the Map object
    function mat = getMaterial(obj,matIdentifier)
      if (obj.db.isKey(matIdentifier))
        mat = obj.db(matIdentifier);
      else
        % Displaying error message if the material class has not been created yet
        error('Material % not present', matIdentifier);
      end
    end
  end

  methods (Access = private)
    % Reading material input file
    function readInputFile(obj, fileName)
      fid = fopen(fileName, 'r');
      while (~feof(fid))
        line = obj.getNewLine(fid);
        if (length(line) == 0)
```

```
34          continue;
35        end
36        % Reading material name (first line)
37        matName = sscanf(line,'%s',1);
38        line = obj.getNewLine(fid);
39        % Reading material identifier (second line)
40        matIdentifier = sscanf(line,'%s',1);
41        data = [];
42        % Reading material data until char = End
43        while (~strcmp(line, 'End'))
44          line = obj.getNewLine(fid);
45          parts = strsplit(line, '%');
46          line = parts{1};
47          if (~strcmp(line, 'End'))
48            data = [data,line];
49          end
50        end
51
52       % Calling the specific material class based on matName
53        switch lower(matName)
54          case 'elastic'
55            obj.db(matIdentifier) = Elastic(data);
56          case 'hypoelastic'
57            obj.db(matIdentifier) = HypoElastic(data);
58          case 'transvelastic'
59            obj.db(matIdentifier) = TransvElastic(data);
60          case 'porousrock'
61            obj.db(matIdentifier) = PorousRock(data);
62          case 'fluid'
63            obj.db(matIdentifier) = Fluid(data);
64          otherwise
65            error('%s not available', matName);
66        end
67      end
68      fclose(fid);
69    end
70
71    % Reading lines from the input file until the end of the file
72    function line = getNewLine(obj, fid)
73      line = fgetl(fid);
74      while (~feof(fid) && length(line) == 0)
75        line = fgetl(fid);
76      end
77    end
78
79  end
80 end
```

Listing A.2: Elastic isotropic material class

```
classdef Elastic < handle
  % ELASTIC ISOTROPIC material class

  properties (Access = private)
    % Elastic modulus
    E = [];
    % Poisson's ratio
    nu = [];
  end

  methods (Access = public)
    % Class constructor method
    function obj = Elastic(inputString)
      % Calling the function to set object properties
      obj.setMaterialParameters(inputString);
    end

    % Material stiffness matrix calculation using the object properties
    function D = getStiffnessMatrix(obj, varargin)
      % Constituent matrix
      D = zeros(6,6);
      D(1,1) = 1-obj.nu;
      D(1,2) = obj.nu;
      D(1,3) = obj.nu;
      D(2,1) = obj.nu;
      D(2,2) = 1-obj.nu;
      D(2,3) = obj.nu;
      D(3,1) = obj.nu;
      D(3,2) = obj.nu;
      D(3,3) = 1-obj.nu;
      D(4,4) = (1-2*obj.nu)/2;
      D(5,5) = (1-2*obj.nu)/2;
      D(6,6) = (1-2*obj.nu)/2;
      D = obj.E/((1+obj.nu)*(1-2*obj.nu))*D;
    end
  end

  methods (Access = private)
    % Assigning material parameters (read inside the class Materials) to object properties
    function setMaterialParameters(obj, inputString)
      words = strsplit(inputString, ' ');
      params = zeros(length(words),1);
      k = 0;
      for i = 1 : length(words)
        if (length(words{i}) > 0)
          k = k + 1;
          params(k) = sscanf(words{i}, '%e');
        end
      end
      % Object properties are assigned with the same order used for material parameters inside
```

```
          the input file
      obj.E = params(1);
      obj.nu = params(2);
    end
  end

end
```

Listing A.3: Transverse isotropic material class

```
classdef TransvElastic < handle
  % ELASTIC TRANSVERSE ISOTROPIC material class

  properties (Access = private)
    % Elastic modulus in the symmetry plane (x,y)
    E_p = [];
    % Elastic modulus in perpendicular direction (z)
    E_z = [];
    % Poisson's ratio in the symmetry plane (x,y)
    nu_p = [];
    % Poisson's ratio in perpendicular direction (z)
    nu_z = [];
  end

  methods (Access = public)
    % Class constructor method
    function obj = TransvElastic(inputString)
      % Calling the function to set object properties
      obj.setMaterialParameters(inputString);
    end

    % Material stiffness matrix calculation using the object properties
    function D = getStiffnessMatrix(obj, varargin)
      % Elastic moduli ratio
      lambda = obj.E_p/obj.E_z;
      % Constituent matrix
      D = zeros(6,6);
      D(1,1) = obj.E_p*(lambda-(obj.nu_z)^2)/((lambda-lambda*obj.nu_p-2*(obj.nu_z)^2)*(1+obj.
          nu_p));
      D(1,2) = obj.E_p*(lambda*obj.nu_p+(obj.nu_z)^2)/((lambda-lambda*obj.nu_p-2*(obj.nu_z)^2)
          *(1+obj.nu_p));
      D(1,3) = obj.E_p*obj.nu_z/(lambda-lambda*obj.nu_p-2*(obj.nu_z)^2);
      D(2,1) = obj.E_p*(lambda*obj.nu_p+(obj.nu_z)^2)/((lambda-lambda*obj.nu_p-2*(obj.nu_z)^2)
          *(1+obj.nu_p));
      D(2,2) = obj.E_p*(lambda-(obj.nu_z)^2)/((lambda-lambda*obj.nu_p-2*(obj.nu_z)^2)*(1+obj.
          nu_p));
      D(2,3) = obj.E_p*obj.nu_z/(lambda-lambda*obj.nu_p-2*(obj.nu_z)^2);
      D(3,1) = obj.E_p*obj.nu_z/(lambda-lambda*obj.nu_p-2*(obj.nu_z)^2);
      D(3,2) = obj.E_p*obj.nu_z/(lambda-lambda*obj.nu_p-2*(obj.nu_z)^2);
      D(3,3) = ((1-obj.nu_p)*obj.E_p)/(lambda-lambda*obj.nu_p-2*(obj.nu_z)^2);
```

```
      D(4,4) = obj.E_p/(2*(1+obj.nu_z));
      D(5,5) = obj.E_p/(2*(1+obj.nu_z));
      D(6,6) = 0.5*(D(1,1)-D(1,2));
      D = abs(D);
    end
  end

  methods (Access = private)
    % Assigning material parameters (read inside the class Materials) to object properties
    function setMaterialParameters(obj, inputString)
      words = strsplit(inputString, ' ');
      params = zeros(length(words),1);
      k = 0;
      for i = 1 : length(words)
        if (length(words{i}) > 0)
          k = k + 1;
          params(k) = sscanf(words{i}, '%e');
        end
      end
      % Object properties are assigned with the same order used for material parameters inside
          the input file
      obj.E_p = params(1);
      obj.E_z = params(2);
      obj.nu_p = params(3);
      obj.nu_z = params(4);
    end
  end

end
```

Listing A.4: Hypoelastic material class

```
classdef HypoElastic < handle
  % HYPOELASTIC ISOTROPIC material class

  properties (Access = private)
    % Poisson's ratio
    nu = [];
    % Coefficients a, b for virgin compressibility
    a = [];
    b = [];
    % Coefficients a1, b1 for unload/reload compressibility
    a1 = [];
    b1 = [];
    % Preconsolidation vertical stress
    szmin = [];
  end

  methods (Access = public)
    % Class constructor method
```

```
19      function obj = HypoElastic(inputString)
20        % Calling the function to set object properties
21        obj.setMaterialParameters(inputString);
22      end
23
24      % Material stiffness matrix calculation using the object properties
25      function D = getStiffnessMatrix(obj, varargin)
26        if (nargin < 2)
27          error('Missing sz in HypoElastic/getStiffnessMatrix');
28        end
29        % vertical stress = first 'getStiffnessMatrix' input value
30        sz = varargin{1};
31        % Constituent matrix
32        D = zeros(6,6);
33        D(1,1) = 1;
34        D(1,2) = obj.nu/(1-obj.nu);
35        D(1,3) = obj.nu/(1-obj.nu);
36        D(2,1) = obj.nu/(1-obj.nu);
37        D(2,2) = 1;
38        D(2,3) = obj.nu/(1-obj.nu);
39        D(3,1) = obj.nu/(1-obj.nu);
40        D(3,2) = obj.nu/(1-obj.nu);
41        D(3,3) = 1;
42        D(4,4) = (1-2*obj.nu)/(2*(1-obj.nu));
43        D(5,5) = (1-2*obj.nu)/(2*(1-obj.nu));
44        D(6,6) = (1-2*obj.nu)/(2*(1-obj.nu));
45        cm = getCompressibility(obj, sz);
46        D = (1/cm)*D;
47      end
48    end
49
50    methods (Access = private)
51      % Assigning material parameters (read inside the class Materials) to object properties
52      function setMaterialParameters(obj, inputString)
53        words = strsplit(inputString, ' ');
54        params = zeros(length(words),1);
55        k = 0;
56        for i = 1 : length(words)
57          if (length(words{i}) > 0)
58            k = k + 1;
59            params(k) = sscanf(words{i}, '%e');
60          end
61        end
62        % Object properties are assigned with the same order used for material parameters inside
              the input file
63        obj.nu = params(1);
64        obj.a = params(2);
65        obj.b = params(3);
66        obj.a1 = params(4);
67        obj.b1 = params(5);
```

```
      obj.szmin = params(6);
    end

    % Compressibility calculation
    function cm = getCompressibility(obj, sz)
       if sz <= obj.szmin
          % Loading path
          cm = (obj.a) * (abs(sz))^(obj.b);
       else
          % Unloading/reloading path
          cm = (obj.a1) * (abs(sz))^(obj.b1);
       end
    end

  end
end
```

Listing A.5: Porous rock class

```
classdef PorousRock < handle
  % POROUS ROCK material class

  properties (Access = private)
    % General properties:
    kx = [];              % Permeability in x
    ky = [];              % Permeability in y
    kz = [];              % Permeability in z
    poro = [];            % Porosity
  end

  methods (Access = public)
    % Class constructor method
    function obj = PorousRock(inputString)
      % Calling the function to set object properties
      obj.setMaterialParameters(inputString);
    end

    % Function to get material porosity
    function poro = getPorosity(obj)
      poro = obj.poro;
    end

    % Function to get material permeability
    function [kx, ky, kz] = getPermeability(obj)
      kx = obj.kx;
      ky = obj.ky;
      kz = obj.kz;
    end
  end

```

```
  methods (Access = private)
    % Assigning material parameters (read inside the class Materials) to object properties
    function setMaterialParameters(obj, inputString)
      words = strsplit(inputString, ' ');
      params = zeros(length(words),1);
      k = 0;
      for i = 1 : length(words)
        if (length(words{i}) > 0)
          k = k + 1;
          params(k) = sscanf(words{i}, '%e');
        end
      end
      % Object properties are assigned with the same order used for material parameters inside
          the input file
      obj.kx = params(1);
      obj.ky = params(2);
      obj.kz = params(3);
      obj.poro = params(4);
    end
  end

end
```

Listing A.6: General fluid class

```
classdef Fluid < handle
  % FLUID general fluid class

  properties (Access = private)
    % General properties:
    gamma = [];             % Fluid specific weight
    beta = [];              % Fluid compressibility
  end

  methods (Access = public)
    % Class constructor method
    function obj = Fluid(inputString)
      % Calling the function setMaterialParameters to set material parameters
      obj.setMaterialParameters(inputString);
    end

    % Function to get material weight
    function gamma = getWeight(obj)
      gamma = obj.gamma;
    end

    % Function to get material compressibility
    function beta = getCompressibility(obj)
      beta = obj.beta;
    end
```

```
 end

  methods (Access = private)
    % Assigning material parameters (read inside the class Materials) to object properties
    function setMaterialParameters(obj, inputString)
      words = strsplit(inputString, ' ');
      params = zeros(length(words),1);
      k = 0;
      for i = 1 : length(words)
        if (length(words{i}) > 0)
          k = k + 1;
          params(k) = sscanf(words{i}, '%e');
        end
      end
      % Object properties are assigned with the same order used for material parameters inside
          the input file
      obj.gamma = params(1);
      obj.beta = params(2);
    end
  end

end
```

Listing A.7: General boundary conditions class

```
classdef Boundaries < handle
  % BOUNDARY CONDITIONS General boundary conditions class

  properties (Access = private)
    % Creation of a Map object
    db = containers.Map;
  end

  methods (Access = public)
    % Class constructor method
    function obj = Boundaries(fileName)
      % Calling the function to read input data from file
      obj.readInputFile(fileName)
    end

    % Check if the BCIdentifier defined by the user is a key of the Map object
    function bcType = getBC(obj, BCIdentifier)
      if (obj.db.isKey(BCIdentifier))
        bcType = obj.db(BCIdentifier);
      else
        % Displaying error message if the boundary class has not been created yet
         error('Boundary condition % not present', BCIdentifier);
      end
    end
  end
```

```
26
27    methods (Access = private)
28      % Reading boundary input file
29      function readInputFile(obj, fileName)
30        fid = fopen(fileName, 'r');
31        while (~feof(fid))
32          line = obj.getNewLine(fid);
33          if (length(line) == 0)
34            continue;
35          end
36          % Reading BC name (first line)
37          BCName = sscanf(line,'%s',1);
38          line = obj.getNewLine(fid);
39          % Reading BC identifier (second line)
40          BCIdentifier = sscanf(line,'%s',1);
41          data = [];
42          % Reading BC data until char = End
43          while (~strcmp(line, 'End'))
44            line = obj.getNewLine(fid);
45            parts = strsplit(line,'%');
46            line = parts{1};
47            if (~strcmp(line, 'End'))
48              data = [data,line];
49            end
50          end
51
52          % Calling the specific boundary condition class based on  BCName
53          switch lower(BCName)
54            case 'nodebc'
55              obj.db(BCIdentifier) = NodeBC(data);
56            otherwise
57              error('%s not available', BCName);
58          end
59         end
60        fclose(fid);
61      end
62
63      % Reading lines from the input file until the end of the file
64      function line = getNewLine(obj, fid)
65        line = fgetl(fid);
66        while (~feof(fid) && length(line) == 0)
67          line = fgetl(fid);
68        end
69      end
70
71    end
72  end
```

Listing A.8: NODAL boundary conditions class

```
classdef NodeBC < handle
  % NODAL BOUNDARY CONDITIONS class
  % Constrained node IDs and the values of the unknowns are defined in the INPUT file.
  % This class returns two vectors, one with the constrained dof(bound_dof) for the stiffness
      matrix and another one with the value of the unknown for each degree of freedom (
      bound_value).

  properties (Access = public)
    % Vector with the total number of constrained nodes for each
    % degree of freedom
    numID = [];
    % IDs of constrained nodes
    ID_x = [];
    ID_y = [];
    ID_z = [];
    % Maximum number of degrees of freedom for each node
    dofmax = 3;
    % Value of the unknown
    value_x = [];
    value_y = [];
    value_z = [];

    % OUTPUT DATA:
    % Indeces of restrained degrees of freedom for the global stiffness matrix, dim = sum(numID)
    bound_dof = [];
    % Values of the unknowns, dim = sum(numID)
    bound_value = [];
  end

  methods (Access = public)
     % Class constructor method
     function obj = NodeBC(inputString)
       % Calling the function to set object properties
       obj.setBCInput(inputString);
     end

     % Generation of the OUTPUT VECTORS
     function NodeBoundary(obj,varargin)
       obj.bound_dof = zeros(sum(obj.numID),1);
       i = 1;
       % Generating bound_dof
       while i <= sum(obj.numID)
          for j = 1 : length(obj.ID_x)
             obj.bound_dof(i) = obj.dofmax*obj.ID_x(j)-2;
             i = i + 1;
          end
          for j = 1 : length(obj.ID_y)
             obj.bound_dof(i) = obj.dofmax*obj.ID_y(j)-1;
             i = i + 1;
          end
```

```
49          for j = 1 : length(obj.ID_z)
50             obj.bound_dof(i) = obj.dofmax*obj.ID_z(j);
51             i = i + 1;
52          end
53       end
54
55       if sum(obj.numID) ~= length(obj.bound_dof)
56          error('Error in nodal boundary conditions: wrong bound_dof size');
57       end
58
59       % Generating bound_value
60       obj.bound_value = zeros(sum(obj.numID),1);
61       i = 1;
62       while i <= sum(obj.numID)
63         for j = 1 : length(obj.value_x)
64            obj.bound_value(i) = obj.value_x(j);
65            i = i + 1;
66         end
67         for j = 1 : length(obj.value_y)
68            obj.bound_value(i) = obj.value_y(j);
69            i = i + 1;
70         end
71         for j = 1 : length(obj.value_z)
72            obj.bound_value(i) = obj.value_z(j);
73            i = i + 1;
74         end
75       end
76
77    end
78
79 end
80
81 methods (Access = private)
82     % Assigning material parameters (read inside the class Boundaries) to object properties
83    function setBCInput(obj, inputString)
84       words = strsplit(inputString, ' ');
85       params = zeros(length(words),1);
86       k = 0;
87       for j = 1 : length(words)
88         if (length(words{j}) > 0)
89            k = k + 1 ;
90            params(k) = sscanf(words{j}, '%e');
91         end
92         % Total number of constrained nodes in the mesh
93         obj.numID = params(1:obj.dofmax);
94       end
95
96       constraint = params(obj.dofmax+1:end);
97       ID = zeros (max(obj.numID),obj.dofmax);
98       value = zeros (max(obj.numID),obj.dofmax);
```

```
99
100       count = 1;
101       for m = 1 : length(obj.numID)
102          for n = 1 : obj.numID(m)
103             if obj.numID(m) ~= 0
104             % Matrix with IDs of constrained nodes (zeros will be ignored)
105             ID(n,m) = constraint(count);
106             % Matrix with the values of the unknowns (zero not related to a specif ID will be
                    ignored)
107             if ID(n,m) == 0
108               break
109             end
110             value(n,m) = constraint(count+1);
111             count = count + 2;
112             end
113          end
114       end
115
116       % Vectors for constrained dof indeces
117       for c = 1 : obj.dofmax
118          [nrow,~] = size(ID);
119          for r = 1 : nrow
120             if (c==1) && (ID(r,c) ~= 0)
121                obj.ID_x(r) = ID(r,c);
122             elseif ID(r,c) == 0
123                 break
124             elseif (c==2) && (ID(r,c) ~= 0)
125                obj.ID_y(r) = ID(r,c);
126             elseif ID(r,c) == 0
127                 break
128             elseif (c==3) && (ID(r,c) ~= 0)
129                obj.ID_z(r) = ID(r,c);
130             elseif ID(r,c) == 0
131                 break
132             end
133          end
134       end
135
136       % Value of the constrained dof
137       for i = 1:length(obj.ID_x)
138         obj.value_x(i) = value(i,1);
139       end
140       for i = 1:length(obj.ID_y)
141         obj.value_y(i) = value(i,2);
142       end
143       for i = 1:length(obj.ID_z)
144         obj.value_z(i) = value(i,3);
145       end
146
147    end
```

```
  end

end
```

Listing A.9: General elements class

```
classdef Elements < handle
  % ELEMENT General element class

  properties (Access = private)
    % Element type
    elemType = [];
    % Element data
    data = [];
  end

  methods (Access = public)
    % Class constructor method
    function obj = Elements(varargin)
      % Calling the function to set element data
      obj.setElementData(varargin)
    end

    % Calling the specific element class based on elemType
    function el = getElement(obj,elemType)
       if elemType == 10
          el = Tetrahedron(obj.data);
       elseif elemType == 12
          el = Hexahedron(obj.data);
       else
          error('Element not available');
       end
    end

  end

  methods (Access = private)
     % Assigns element data (properties of the 'Mesh' object)
     function obj = setElementData(obj,varargin)
        obj.data = varargin{1};
     end
  end

end
```

Listing A.10: Tetrahedron element class

```
classdef Tetrahedron < handle
 % TETRAHEDRON element class

```

```
4   properties (Access = public)
5     % INPUT
6     dofmax = 3;
7     % Number of faces of the element
8     nFace = 4;
9     % Number of nodes of the element
10    nNode = [];
11    % Nodes coordinates
12    nodeCoords = [];
13    % Total number of elements in the mesh
14    nElem = [];
15    % Elements nodes sequence
16    elemTopol = [];
17    % Surfaces nodes sequence
18    surfTopol = [];
19
20    % OUTPUT
21    % Elements volume
22    Vol = [];
23    % Surfaces Area
24    Area = [];
25    % Basis function matrix
26    N = [];
27    % Basis function derivatives matrix
28    B = [];
29  end
30
31  methods (Access = public)
32    % Class constructor method
33    function obj = Tetrahedron(data)
34       % Calling the function to set element data
35       obj.setElementData(data);
36    end
37
38    % Basis function coefficients calculation
39    function getBasisF(obj)
40       obj.N = zeros(3*obj.nElem,obj.nNode);
41
42       % Linear tetrahedron
43       if obj.nNode == 4
44
45          for el = 1:obj.nElem
46            i = obj.elemTopol(el,1);
47            j = obj.elemTopol(el,2);
48            m = obj.elemTopol(el,3);
49            p = obj.elemTopol(el,4);
50
51            b(1) = -det([1 obj.nodeCoords(j,2) obj.nodeCoords(j,3);
52                         1 obj.nodeCoords(m,2) obj.nodeCoords(m,3);
53                         1 obj.nodeCoords(p,2) obj.nodeCoords(p,3);]);
```

```
54            c(1) = det([1 obj.nodeCoords(j,1) obj.nodeCoords(j,3);
55                        1 obj.nodeCoords(m,1) obj.nodeCoords(m,3);
56                        1 obj.nodeCoords(p,1) obj.nodeCoords(p,3);]);
57            d(1) = -det([1 obj.nodeCoords(j,1) obj.nodeCoords(j,2);
58                         1 obj.nodeCoords(m,1) obj.nodeCoords(m,2);
59                         1 obj.nodeCoords(p,1) obj.nodeCoords(p,2);]);
60
61            b(2) = det([1 obj.nodeCoords(i,2) obj.nodeCoords(i,3);
62                        1 obj.nodeCoords(m,2) obj.nodeCoords(m,3);
63                        1 obj.nodeCoords(p,2) obj.nodeCoords(p,3);]);
64            c(2) = -det([1 obj.nodeCoords(i,1) obj.nodeCoords(i,3);
65                         1 obj.nodeCoords(m,1) obj.nodeCoords(m,3);
66                         1 obj.nodeCoords(p,1) obj.nodeCoords(p,3);]);
67            d(2) = det([1 obj.nodeCoords(i,1) obj.nodeCoords(i,2);
68                        1 obj.nodeCoords(m,1) obj.nodeCoords(m,2);
69                        1 obj.nodeCoords(p,1) obj.nodeCoords(p,2);]);
70
71            b(3) = -det([1 obj.nodeCoords(i,2) obj.nodeCoords(i,3);
72                         1 obj.nodeCoords(j,2) obj.nodeCoords(j,3);
73                         1 obj.nodeCoords(p,2) obj.nodeCoords(p,3);]);
74            c(3) = det([1 obj.nodeCoords(i,1) obj.nodeCoords(i,3);
75                        1 obj.nodeCoords(j,1) obj.nodeCoords(j,3);
76                        1 obj.nodeCoords(p,1) obj.nodeCoords(p,3);]);
77            d(3) = -det([1 obj.nodeCoords(i,1) obj.nodeCoords(i,2);
78                         1 obj.nodeCoords(j,1) obj.nodeCoords(j,2);
79                         1 obj.nodeCoords(p,1) obj.nodeCoords(p,2);]);
80
81            b(4) = det([1 obj.nodeCoords(i,2) obj.nodeCoords(i,3);
82                        1 obj.nodeCoords(j,2) obj.nodeCoords(j,3);
83                        1 obj.nodeCoords(m,2) obj.nodeCoords(m,3);]);
84            c(4) = -det([1 obj.nodeCoords(i,1) obj.nodeCoords(i,3);
85                         1 obj.nodeCoords(j,1) obj.nodeCoords(j,3);
86                         1 obj.nodeCoords(m,1) obj.nodeCoords(m,3);]);
87            d(4) = det([1 obj.nodeCoords(i,1) obj.nodeCoords(i,2);
88                        1 obj.nodeCoords(j,1) obj.nodeCoords(j,2);
89                        1 obj.nodeCoords(m,1) obj.nodeCoords(m,2);]);
90
91            obj.N(3*el-2,:) = b;
92            obj.N(3*el-1,:) = c;
93            obj.N(3*el,:) = d;
94          end
95
96        % Quadratic tetrahedron
97        elseif obj.nNode == 10
98            error('Element not available');
99        end
100   end
101
102   % Elements volume calculation
103   function getVolume(obj)
```

```
104         obj.Vol = zeros(obj.nElem,1);
105         for el = 1:obj.nElem
106             i = obj.elemTopol(el,1);
107             j = obj.elemTopol(el,2);
108             m = obj.elemTopol(el,3);
109             p = obj.elemTopol(el,4);
110 
111             obj.Vol(el) = (det([1 obj.nodeCoords(i,1) obj.nodeCoords(i,2) obj.nodeCoords(i,3);
112                                 1 obj.nodeCoords(j,1) obj.nodeCoords(j,2) obj.nodeCoords(j,3);
113                                 1 obj.nodeCoords(m,1) obj.nodeCoords(m,2) obj.nodeCoords(m,3);
114                                 1 obj.nodeCoords(p,1) obj.nodeCoords(p,2) obj.nodeCoords(p,3)]))
                                     /6;
115         end
116     end
117 
118     % Basis function derivatives calculation
119     function getDerivatives(obj)
120        obj.B = zeros(2*obj.dofmax*obj.nElem,obj.nNode*obj.dofmax);
121 
122        % Linear tetrahedron
123        if obj.nNode == 4
124 
125          for el = 1: obj.nElem
126           b = obj.N(3*el-2,:);
127           c = obj.N(3*el-1,:);
128           d = obj.N(3*el,:);
129 
130              B1 = [b(1)   0     0;
131                    0    c(1)    0;
132                    0    0    d(1);
133                    c(1) b(1)    0;
134                    0    d(1) c(1);
135                    d(1) 0    b(1)];
136 
137              B2 = [b(2)   0     0;
138                    0    c(2)    0;
139                    0    0    d(2);
140                    c(2) b(2)    0;
141                    0    d(2) c(2);
142                    d(2) 0    b(2)];
143 
144              B3 = [b(3)   0     0;
145                    0    c(3)    0;
146                    0    0    d(3);
147                    c(3) b(3)    0;
148                    0    d(3) c(3);
149                    d(3) 0    b(3)];
150 
151              B4 = [b(4)   0     0;
152                    0    c(4)    0;
```

```
                0    0    d(4);
                c(4) b(4)    0;
                0    d(4) c(4);
                d(4) 0    b(4)];

        obj.B (6*el-5:6*el,:)  = (1/(6*obj.Vol(el)))* [B1 B2 B3 B4];
        end

      % Quadratic tetrahedron
      elseif obj.nNode == 10
          error('Element not available');
      end
    end

    % Elements surfaces area calculation
    function getArea(obj)
      [nrow,ncol] = size(obj.surfTopol);
      obj.Area = zeros(nrow,1);
         for f = 1:nrow
             for ind = 1:ncol
                node = obj.surfTopol(f,ind);
                coord(ind,:) = obj.nodeCoords(node,:);
             end
             n1 = coord(1,:);
             n2 = coord(2,:);
             n3 = coord(3,:);

             v1 = [n2(1)-n1(1),n2(2)-n1(2),n2(3)-n1(3)];
             v2 = [n3(1)-n1(1),n3(2)-n1(2),n3(3)-n1(3)];

             prodvett = cross(v1,v2);
             norma = norm(prodvett);
             obj.Area(f) = 0.5*norma;
         end
     end

end

methods (Access = private)
    % Assigning the object properties from mesh data
    function setElementData(obj,data)
        nNode = data{1};
        obj.nNode = nNode(1,1);
        obj.nodeCoords = data{2};
        obj.nElem = data{3};
        obj.elemTopol = data{4};
        obj.surfTopol = data{5};
    end
end
```

```
end
```

Listing A.11: Hexahedron element class

```
classdef Hexahedron < handle
  % HEXAHEDRON element class

  properties (Access = public)
    % INPUT
    % Degrees of freedom for each node
    dofmax = 3;
    % 3D element
    dim = 3;
    % Number of faces of the element
    nFace = 6;
    % Number of nodes of the element
    nNode = [];
    % Nodes coordinates
    nodeCoords = [];
    % Total number of elements in the mesh
    nElem = [];
    % Elements nodes sequence
    elemTopol = [];
    % Cell centroid
    cellCentroid = [];
    % Surfaces nodes sequence
    surfTopol = [];

    % OUTPUT
    % Elements side lenghts
    S_lenght = [];
    % Elements volume
    Vol = [];
    % Surfaces Area
    Area = [];
    % Basis functions matrix
    N = [];
    % Basis function derivatives matrix
    B = [];
    % Jacobian matrix
    J = [];
    % Basis function derivatives with respect to natural coordinates
    derMatrix = [];
  end

  methods (Access = public)
    % Class constructor method
    function obj = Hexahedron(data)
       % Calling the function to set element data
       obj.setElementData(data);
```

```
47      end
48
49      % Elements side lenghts calculation
50      function getSide(obj)
51         obj.S_lenght = zeros(obj.nElem,3);
52         for el = 1 : obj.nElem   % element row in element topology
53             a = 0;
54             b = 0;
55             c = 0;
56             j = obj.elemTopol(el,1); % first node id in element topology
57             for i = 2 :obj.nNode
58                m = obj.elemTopol(el,i);
59                if (obj.nodeCoords(j,1)) ~= (obj.nodeCoords(m,1)) && a==0
60                a = abs(obj.nodeCoords(m,1)-obj.nodeCoords(j,1));
61                elseif (obj.nodeCoords(j,2)) ~= (obj.nodeCoords(m,2)) && b==0
62                b = abs(obj.nodeCoords(m,2)-obj.nodeCoords(j,2));
63                elseif (obj.nodeCoords(j,3)) ~= (obj.nodeCoords(m,3)) && c==0
64                c = abs(obj.nodeCoords(m,3)-obj.nodeCoords(j,3));
65                end
66             end
67             if a~=0 && b~=0 && c~=0
68               obj.S_lenght(el,:) = [a b c];
69             else
70               error('Error computing element side length')
71             end
72         end
73
74      end
75
76      % Re-ordering the element topology based on the natural coordinate system
77      function obj = reOrderTopol(obj)
78        old_topol = obj.elemTopol;
79        new_topol = zeros(obj.nElem,obj.nNode);
80        for el = 1:obj.nElem
81            for i = 1:obj.nNode
82               node = old_topol(el,i);
83               x_coords(1,i) = obj.nodeCoords(node,1);
84               y_coords(1,i) = obj.nodeCoords(node,2);
85               z_coords(1,i) = obj.nodeCoords(node,3);
86            end
87            x_min = min(x_coords);
88            y_min = min(y_coords);
89            z_min = min(z_coords);
90
91            for i = 1:obj.nNode
92                node = old_topol(el,i);
93                c1 = obj.nodeCoords(node,1);
94                c2 = obj.nodeCoords(node,2);
95                c3 = obj.nodeCoords(node,3);
96                if c1 == x_min && c2 == y_min && c3 == z_min
```

```
97                  new_topol(el,1) = node;
98              elseif c1 ~= x_min && c2 == y_min && c3 == z_min
99                  new_topol(el,2) = node;
100             elseif c1 ~= x_min && c2 ~= y_min && c3 == z_min
101                 new_topol(el,3) = node;
102             elseif c1 == x_min && c2 ~= y_min && c3 == z_min
103                 new_topol(el,4) = node;
104             elseif c1 == x_min && c2 == y_min && c3 ~= z_min
105                 new_topol(el,5) = node;
106             elseif c1 ~= x_min && c2 == y_min && c3 ~= z_min
107                 new_topol(el,6) = node;
108             elseif c1 ~= x_min && c2 ~= y_min && c3 ~= z_min
109                 new_topol(el,7) = node;
110             elseif c1 == x_min && c2 ~= y_min && c3 ~= z_min
111                 new_topol(el,8) = node;
112             end
113         end
114         obj.elemTopol(el,:) = new_topol(el,:);
115     end
116 end
117
118 % Basis function matrix calculation
119 function getBasisF(obj)
120     obj.N = zeros(obj.dofmax*obj.nElem,obj.dofmax*obj.nNode);
121
122     % Linear hexahedron
123     if obj.nNode == 8
124
125         for el = 1:obj.nElem
126           for j = 1:obj.nNode
127             nod = obj.elemTopol(el,j);
128
129             s = (obj.nodeCoords(nod,1)-obj.cellCentroid(el,1))/(0.5*obj.Slenght(el,1));
130             t = (obj.nodeCoords(nod,2)-obj.cellCentroid(el,2))/(0.5*obj.Slenght(el,2));
131             p = (obj.nodeCoords(nod,3)-obj.cellCentroid(el,3))/(0.5*obj.Slenght(el,3));
132
133             N1 = 1/8*(1-s)*(1-t)*(1-p);
134             N2 = 1/8*(1+s)*(1-t)*(1-p);
135             N3 = 1/8*(1+s)*(1+t)*(1-p);
136             N4 = 1/8*(1-s)*(1+t)*(1-p);
137             N5 = 1/8*(1-s)*(1-t)*(1+p);
138             N6 = 1/8*(1+s)*(1-t)*(1+p);
139             N7 = 1/8*(1+s)*(1+t)*(1+p);
140             N8 = 1/8*(1-s)*(1+t)*(1+p);
141
142             totN = N1+N2+N3+N4+N5+N6+N7+N8;
143             if totN ~= 1
144                 error('Shape functions must have local support')
145             end
146
```

```
147             obj.N(3*el-2:3*el,3*j-2:3*j) = [totN 0 0;
148                                             0 totN 0;
149                                             0 0 totN];
150           end
151         end
152
153       % Quadratic hexahedron
154       elseif obj.nNode == 20
155           error('Element not available');
156       end
157   end
158
159   % Jacobian matrix calculation
160   function obj = getJacobian(obj,varargin)
161      el = varargin{1};
162
163      % Linear hexahedron
164      if obj.nNode == 8
165
166         s = varargin{2};
167         t = varargin{3};
168         p = varargin{4};
169         for n = 1:obj.nNode
170            if n == 1 %first node of the element
171               dNs = -1/8*(1-t)*(1-p);
172               dNt = -1/8*(1-s)*(1-p);
173               dNp = -1/8*(1-s)*(1-t);
174
175            elseif n == 2 %second node of the element
176               dNs = 1/8*(1-t)*(1-p);
177               dNt = -1/8*(1+s)*(1-p);
178               dNp = -1/8*(1+s)*(1-t);
179
180            elseif n == 3 %third node of the element
181               dNs = 1/8*(1+t)*(1-p);
182               dNt = 1/8*(1+s)*(1-p);
183               dNp = -1/8*(1+s)*(1+t);
184
185            elseif n == 4 %fourth node of the element
186               dNs = -1/8*(1+t)*(1-p);
187               dNt = 1/8*(1-s)*(1-p);
188               dNp = -1/8*(1-s)*(1+t);
189
190            elseif n == 5 %fifth node of the element
191               dNs = -1/8*(1-t)*(1+p);
192               dNt = -1/8*(1-s)*(1+p);
193               dNp = 1/8*(1-s)*(1-t);
194
195            elseif n == 6 %sixth node of the element
196               dNs = 1/8*(1-t)*(1+p);
```

```
197                dNt = -1/8*(1+s)*(1+p);
198                dNp = 1/8*(1+s)*(1-t);
199
200             elseif n == 7 %seventh node of the element
201                dNs = 1/8*(1+t)*(1+p);
202                dNt = 1/8*(1+s)*(1+p);
203                dNp = 1/8*(1+s)*(1+t);
204
205             elseif n == 8 %eighth node of the element
206                dNs = -1/8*(1+t)*(1+p);
207                dNt = 1/8*(1-s)*(1+p);
208                dNp = 1/8*(1-s)*(1+t);
209             end
210             dN = [dNs; dNt; dNp];
211             obj.derMatrix (:,n) = dN;
212             nod = obj.elemTopol(el,n);
213             coordMatrix(n,:) = obj.nodeCoords(nod,:);
214          end
215          obj.J = obj.derMatrix*coordMatrix;
216
217       % Quadratic hexahedron
218       elseif obj.nNode == 20
219             error('Element not available');
220       end
221    end
222
223    % Basis function derivatives matrix calculation
224    function obj = getDerivatives(obj,varargin)
225
226       % Linear hexahedron
227       if obj.nNode == 8
228          invJ = inv(obj.J);
229
230          for n = 1:obj.nNode
231             if n == 1 %first node of the element
232               b = invJ(1,:)*obj.derMatrix(:,n);
233               c = invJ(2,:)*obj.derMatrix(:,n);
234               d = invJ(3,:)*obj.derMatrix(:,n);
235
236             elseif n == 2 %second node of the element
237               b = invJ(1,:)*obj.derMatrix(:,n);
238               c = invJ(2,:)*obj.derMatrix(:,n);
239               d = invJ(3,:)*obj.derMatrix(:,n);
240
241             elseif n == 3 %third node of the element
242               b = invJ(1,:)*obj.derMatrix(:,n);
243               c = invJ(2,:)*obj.derMatrix(:,n);
244               d = invJ(3,:)*obj.derMatrix(:,n);
245
246             elseif n == 4 %fourth node of the element
```

```
247             b = invJ(1,:)*obj.derMatrix(:,n);
248             c = invJ(2,:)*obj.derMatrix(:,n);
249             d = invJ(3,:)*obj.derMatrix(:,n);
250
251           elseif n == 5 %fifth node of the element
252             b = invJ(1,:)*obj.derMatrix(:,n);
253             c = invJ(2,:)*obj.derMatrix(:,n);
254             d = invJ(3,:)*obj.derMatrix(:,n);
255
256           elseif n == 6 %sixth node of the element
257             b = invJ(1,:)*obj.derMatrix(:,n);
258             c = invJ(2,:)*obj.derMatrix(:,n);
259             d = invJ(3,:)*obj.derMatrix(:,n);
260
261           elseif n == 7 %seventh node of the element
262             b = invJ(1,:)*obj.derMatrix(:,n);
263             c = invJ(2,:)*obj.derMatrix(:,n);
264             d = invJ(3,:)*obj.derMatrix(:,n);
265
266           elseif n == 8 %eighth node of the element
267             b = invJ(1,:)*obj.derMatrix(:,n);
268             c = invJ(2,:)*obj.derMatrix(:,n);
269             d = invJ(3,:)*obj.derMatrix(:,n);
270           end
271           obj.B(:,3*n-2:3*n) = [b    0    0;
272                                 0    c    0;
273                                 0    0    d;
274                                 c    b    0;
275                                 0    d    c;
276                                 d    0    b];
277       end
278
279     % Quadratic hexahedron
280     elseif obj.nNode == 20
281           error('Element not available');
282     end
283   end
284
285   % Area calculation of the elements surfaces
286   function getArea(obj)
287     [nrow,ncol] = size(obj.surfTopol);
288     obj.Area = zeros(nrow,1);
289     for s = 1:nrow
290         for i = 1:ncol
291            node = obj.surfTopol(s,i);
292            coords(i,:) = obj.nodeCoords(node,:);
293         end
294         L = zeros (1,obj.dofmax);
295         for k = 1:obj.dofmax
296             for i = 1:ncol
```

```
                    if i == ncol
                        break
                    elseif coords(i,k) ~= coords(i+1,k)
                        L(k) = abs(coords(i,k) - coords(i+1,k));
                        if L(k) ~= 0
                            break
                        end
                    end
                end
            end
            L(L==0) = [];
            obj.Area(s) = L(1)*L(2);
        end
    end

    % Elements volume calculation
    function getVolume(obj)
        obj.Vol = zeros(obj.nElem,1);
        for el = 1:obj.nElem
            a = obj.S_lenght(el,1);
            b = obj.S_lenght(el,2);
            c = obj.S_lenght(el,3);
            obj.Vol(el) = a*b*c;
        end
    end

  end

  methods (Access = private)
     % Function that set the object properties from mesh data
     function setElementData(obj,data)
        nNode = data{1};
        obj.nNode = nNode(1,1);
        obj.nodeCoords = data{2};
        obj.nElem = data{3};
        obj.elemTopol = data{4};
        obj.surfTopol = data{5};
        obj.cellCentroid = data{6};
     end
  end

end
```

Listing A.12: Assembly stiffness matrix function

```
function K = assemblyK(nTotNode, elemMAT, mat, element, varargin)
 % Function for global stiffness matrix assembly with no boundary conditions

  % Global stiffness matrix dimension
  Kdim = element.dofmax*nTotNode;
```

```
6
7   % Global stiffness matrix memory allocation
8   K = sparse(Kdim,Kdim,0);
9
10  for el = 1: element.nElem
11    % Get the right material stiffness for each element
12    for i = 1 : length(mat.db.keys)
13      if elemMAT(el) == i
14         var = mat.db.keys;
15         matIdentifier = var(i);
16         matIdentifier = char(matIdentifier);
17      end
18    end
19    % Material stiffness matrix
20    if nargin > 4
21       sz_vec = varargin{1};
22       sz = sz_vec(el);
23       D = mat.getMaterial(matIdentifier).getStiffnessMatrix(sz);
24    else
25       D = mat.getMaterial(matIdentifier).getStiffnessMatrix();
26    end
27
28    % Assembly local stiffness matrices for each element
29    Bt = (element.B(6*el-5:6*el,:))';
30    k = element.Vol(el)*Bt*D*element.B(6*el-5:6*el,:);
31
32    % Number of degrees of freedom for each node
33    dof = length(k)/element.nNode;
34
35    % Assembly local stiffness matrix terms inside global stiffness matrix
36    for i_loc = 1:element.nNode
37      i_glob = element.elemTopol(el,i_loc);
38      for j_loc = 1:element.nNode
39        j_glob = element.elemTopol(el,j_loc);
40
41        ii_global = (i_glob-1)*element.dofmax+1:(i_glob-1)*element.dofmax+3;
42        jj_global = (j_glob-1)*element.dofmax+1:(j_glob-1)*element.dofmax+3;
43
44        ii_local = (i_loc-1)*dof+1:i_loc*dof;
45        jj_local = (j_loc-1)*dof+1:j_loc*dof;
46
47        K(ii_global,jj_global) = K(ii_global,jj_global)+k(ii_local,jj_local);
48      end
49    end
50
51  end
52
53  % Singular matrix check
54  for i = 1 : Kdim
55     if K(i,i) == 0
```

```
        error ('Singular matrix')
    end
  end

end
```

Listing A.13: Assembly stiffness matrix function with Gauss integration rule

```
function K = assemblyKGauss(nTotNode, elemMAT, mat, element, varargin)
% Function for global stiffness matrix assembly with no boundary conditions

 % Global stiffness matrix dimension
 Kdim = element.dofmax*nTotNode;

 % Global stiffness matrix memory allocation
 K = sparse(Kdim,Kdim,0);

  % Assembly local stiffness matrices for each element
  for el = 1: element.nElem
    k = zeros(element.dofmax*element.nNode);

    % Gauss integration points coordinates and weights
    [gCoord,gWeight] = gaussValue(element.nNode,element.dim);
    [~,ncol] = size(gCoord);

    % Sum of stiffness matrices calculated in each Gauss points
    % j = n-th Gauss point
    pGauss = 1;
    while pGauss <= ncol
       s = gCoord(1,:);
       t = gCoord(2,:);
       p = gCoord(3,:);
       s = s(pGauss);
       t = t(pGauss);
       p = p(pGauss);

       element.getJacobian(el,s,t,p);
       element.getDerivatives(el,s,t,p);
       Bt = (element.B)';

       % Get the right material stiffness for each element
       for i = 1 : length(mat.db.keys)
          if elemMAT(el) == i
             var = mat.db.keys;
             matIdentifier = var(i);
             matIdentifier = char(matIdentifier);
          end
       end
       % Material stiffness matrix
       if nargin > 4
```

```
43          sz_vec = varargin{1};
44          sz = sz_vec(el);
45          D = mat.getMaterial(matIdentifier).getStiffnessMatrix(sz);
46        else
47          D = mat.getMaterial(matIdentifier).getStiffnessMatrix();
48        end
49
50        % Calculate stiffness contribution in n-th Gauss point
51        for d = 1:element.dim
52          W = gWeight(pGauss,d);
53          k_pGauss = Bt*D*(element.B)*(det(element.J))*W;
54        end
55
56        % Assembly the element local stiffness matrix
57        k = k + k_pGauss;
58        pGauss = pGauss+1;
59      end
60
61      % Number of degrees of freedom for each node
62      dof = length(k)/element.nNode;
63
64      % Assembly local stiffness matrix terms inside global stiffness matrix
65      for i_loc = 1:element.nNode
66          i_glob = element.elemTopol(el,i_loc);
67
68        for j_loc = 1:element.nNode
69            j_glob = element.elemTopol(el,j_loc);
70
71            ii_global = (i_glob-1)*element.dofmax+1:(i_glob-1)*element.dofmax+3;
72            jj_global = (j_glob-1)*element.dofmax+1:(j_glob-1)*element.dofmax+3;
73
74            ii_local = (i_loc-1)*dof+1:i_loc*dof;
75            jj_local = (j_loc-1)*dof+1:j_loc*dof;
76
77            K(ii_global,jj_global) = K(ii_global,jj_global)+k(ii_local,jj_local);
78        end
79      end
80    end
81
82    % Singular matrix check
83    for i = 1 : Kdim
84         if K(i,i) == 0
85             error ('Singular matrix')
86         end
87    end
88
89  end
```

Listing A.14: Impose boundary conditions function

```
function [K,F] = imposeBC(nTotNode, K, dir, neu)
% Function for boundary conditions imposition in global stiffness matrix

  % 1) Known term
  F = zeros(neu.dofmax*nTotNode,1);
  for i = 1:length(K)
    for j = 1:length(neu.bound_dof)
      if i == neu.bound_dof(j)
         F(i) = F(i) + neu.bound_value(j);
      end
    end
  end


  % 2) Imposing boundary conditions (penalty method)
  % Dirichlet value R
  R = max(K, [], 'all');
  for i = 1:length(dir.bound_dof)
    j = dir.bound_dof(i);
    % Known term vector
    F(j) = dir.bound_value(i)*(R*10^10);
    for col = 1:length(K)
       for row = 1:length(K)
          if row == j
              if row == col
                 K(row,col) = R*10^10;
              end
          end
       end
    end
  end


end
```

Bibliografia

A. Burghignoli. *Lezioni di Meccanica delle Terre*. E.S.A. Editrice, 1985.

G. Gambolati and M. Ferronato. *Lezioni di Metodi Numerici per l'Ingegneria*. Edizioni Progetto, third edition, 2015.

D.L. Logan. *A First Course in the Finite Element Method*. CENGAGE Learning, fifth edition, 2012.

C. e Salomoni V. Majorana. *Scienza delle Costruzioni*. Città Studi EDIZIONI, fifth edition, 2015.

E. Oñate. *Structural Analysis with the Finite Element Method. Linear Statics. Volume 1. Basis and Solids*. Springer, first edition, 2009.

N. Spiezia, M. Ferronato, C. Janna, and P. Teatini. A two-invariant pseudoelastic model for reservoir compaction. *Wiley*, 2017.

O.C. Zienkiewicz and R.L. Taylor. *The Finite Element Method*. Butterwotrh-Heinemann, fifth edition, 2000.