

LAUREA MAGISTRALE
IN INGEGNERIA MATEMATICA

Progetto per il corso di
Programmazione Avanzata per il Calcolo Scientifico.



**Implementazione in LifeV dell'algoritmo di
Riduzione Gerarchica di Modello**

Progetto svolto da:
Matteo Carlo Maria Aletti
Matr. 783045
Andrea Bortolossi
Matr. 783023

Anno Accademico 2012–2013

Indice

1	Introduzione	2
1.1	Nozioni base	2
1.2	Forma matriciale	3
1.3	Educared Basis	6
1.4	Ipotesi	7
2	Descrizione classi	9
2.1	Modalspace	9
2.1.1	Costruzione e setting	9
2.1.2	Metodi di calcolo	12
2.1.3	EigensProvider()	14
2.2	Basis1dAbstract	16
2.3	HiModAssembler	18
2.3.1	I metodi	19
3	Risultati	24

Capitolo 1

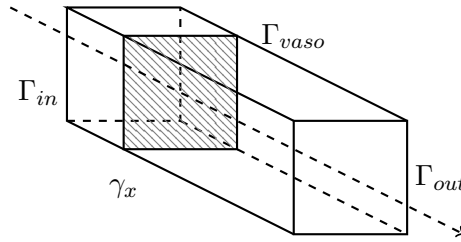
Introduzione

1.1 Nozioni base

L'obiettivo di questo progetto è l'implementazione in **LifeV** di un risolutore ADR 3D, basato sulla tecnica di Riduzione Gerarchica di Modello (Hierarchical Model Reduction - HiMod). Consideriamo il seguente problema stazionario ai limiti:

$$\begin{cases} -\mu\Delta u + \mathbf{b} \cdot \nabla u + \sigma u = f & \text{in } \Omega \\ u = u_{in} & \text{su } \Gamma_{in} \\ \frac{\partial u}{\partial \mathbf{n}} = 0 & \text{su } \Gamma_{out} \\ u = 0 & \text{su } \Gamma_{vaso} \end{cases} \quad (1.1)$$

$$\Omega = \bigcup_{x \in \Omega_{1D}} \gamma_x \quad (1.2)$$



Immaginiamo di suddividere il dominio Ω in slice poste trasversalmente alla direzione longitudinale. Ognuna di queste slice verrà indicata con Ω_{1D} .

Lungo γ_x vengono utilizzate funzioni spaziali differenti rispetto a quelle utilizzate lungo Ω_{1D} . Si consideri infatti per Ω_{1D} , lo spazio funzionale $V_{1D} = H_{\Gamma_{in}}^1(\Omega_{1D})$, mentre sulla generica γ_x si introducano le basi modali $\{\varphi_k(y, z)\}$ ortonormali in $L^2(\gamma_x)$, con $k \in \mathbb{N}$. Quest'ultime definiscono su γ_x lo spazio funzionale $V_{\gamma_x} := \text{span}\{\varphi_k\}$.

Definiamo ora il sottospazio generato solo dai primi m modi ovvero $V_{\gamma_x}^m := \text{span}\{\varphi_1, \dots, \varphi_m\}$ e combiniamolo con V_{1D} , il risultato di tale operazione è il seguente spazio ridotto:

$$V_m := \left\{ v_m(x, y, z) = \sum_{k=1}^m \varphi_k(y, z) \tilde{v}_k(x), \text{ con } \tilde{v}_k \in V_{1D} \right\} \quad (1.3)$$

Questo è l'ambiente funzionale in cui opera il metodo HiMod. L'ortogonalità in $L^2(\gamma_x)$ implica che i coefficienti \tilde{v}_k in (1.3) sono il risultato del seguente prodotto scalare per $k = 1, \dots, m$:

$$\tilde{v}_k(x) = \int_{\gamma_x} \varphi_k(y, z) v_m(x, y, z) dydz$$

La convergenza di una soluzione u_m tale che soddisfi il problema (1.1) è stata dimostrata ([citazione](#)). Valgono inoltre le seguenti proprietà:

- $V_m \subset V \forall m \in \mathbb{N}$, ossia che lo spazio ridotto V_m è **conforme** in V ;
- $\lim_{m \rightarrow +\infty} \left(\inf_{v_m \in V_m} \|v - v_m\| \right) = 0$ per ogni $v \in V$, ossia che vale la **proprietà di approssimazione** di V_m rispetto a V ;

Le proprietà continuano a valere anche nel caso di dato di Dirichlet non omogeneo sulle pareti del vaso (?).

1.2 Forma matriciale

Per ogni $m \in \mathbb{N}$ consideriamo il seguente problema ridotto di (1.1), trovare $u_m \in V_m$ tale che $\forall v_m \in V_m$:

$$\int_{\Omega} (\mu \nabla u_m \nabla v_m + \mathbf{b} \nabla u_m v_m + \sigma u_m v_m) d\Omega = \int_{\Omega} f v dx dy \quad (1.4)$$

Adoperiamo l'espansione tramite i coefficienti di Fourier della $u_m(x, y, z) = \sum_{j=k}^m \tilde{u}_j(x) \varphi_j(y, z)$ dove:

$$\tilde{u}_j(x) = \int_{\gamma(x)} u_m(x, y, z) \varphi_j(y, z) dydz$$

consideriamo inoltre le funzioni test $v_m = \vartheta(x) \varphi_k(y, z)$ con $\vartheta(x) \in V_{1D}$ e $k = 1, \dots, m$. Il problema assume la seguente forma:

$$\begin{aligned}
& \sum_{j=1}^m \left[\int_{\Omega} \mu \nabla(\tilde{u}_j(x) \varphi_j(y, z)) \nabla(\vartheta(x) \varphi_k(y, z)) \, dx dy dz \right. \\
& + \int_{\Omega} \mathbf{b} \nabla(\tilde{u}_j(x) \varphi_j(y, z) \vartheta(x) \varphi_k(y, z)) \, dx dy dz \\
& \left. + \int_{\Omega} \sigma \tilde{u}_j(x) \varphi_j(y, z) \vartheta(x) \varphi_k(y, z) \, dx dy dz \right] \\
& = \int_{\Omega} f \vartheta(x) \varphi_k(y, z) \, dx dy dz
\end{aligned} \tag{1.5}$$

Svolgendo l'operatore gradiente si ottiene:

$$\begin{aligned}
& \sum_{j=1}^m \left[\int_{\Omega} \mu (\partial_x \tilde{u}_j \partial_x \vartheta \varphi_j \varphi_k + \tilde{u}_j \vartheta \partial_y \varphi_j \partial_y \varphi_k + \tilde{u}_j \vartheta \partial_z \varphi_j \partial_z \varphi_k) \, dx dy dz \right. \\
& + \int_{\Omega} (b_1 \partial_x \tilde{u}_j \varphi_j + b_2 \tilde{u}_j \partial_y \varphi_j + b_3 \tilde{u}_j \partial_z \varphi_j) \vartheta \varphi_k \, dx dy dz \\
& \left. + \int_{\Omega} \sigma \tilde{u}_j \vartheta \varphi_j \varphi_k \, dx dy dz \right] \\
& = \int_{\Omega} f \vartheta \varphi_k \, dx dy dz \tag{1.6}
\end{aligned}$$

Definiamo con N il numero di nodi scelti, uniformemente distribuiti lungo Ω_{1D} . La partizione T_h , costruita lungo la fibra di supporto 1D, avrà un passo spaziale $h = |\Omega_{1D}|/(N-1)$. Introduciamo lo spazio agli elementi finiti lungo Ω_{1D} definito come segue

$$X_h^r = \{ \psi_h \in C^0(\Omega_{1D}) : \psi_h|_K \in \mathbb{P}_r, \forall K \in T_h \}^1 \tag{1.7}$$

Possiamo quindi esprimere i coefficienti di Fourier nel seguente modo:

$$\tilde{u}_j(x) = \sum_{s=1}^N u_{js} \psi_s(x) \tag{1.8}$$

Otteniamo la formulazione matriciale del nostro problema ovvero, trovare $\mathbf{u} \in \mathbb{R}^{N \times m}$ tale che $\forall \psi_l$ e $\forall \varphi_k$, con $l = 1, \dots, N$ e $k = 1, \dots, m$ si ha che:

¹Il codice è stato scritto in modo da utilizzare polinomi di primo grado, la generalizzazione non rientrava negli obbiettivi primari del progetto

$$\begin{aligned}
& \sum_{j=1}^m \sum_{s=1}^N u_{js} \left[\int_{\Omega} \mu (\partial_x \psi_s \partial_x \psi_l \varphi_j \varphi_k + \psi_s \psi_l \partial_y \varphi_j \partial_y \varphi_k + \psi_s \psi_l \partial_z \varphi_j \partial_z \varphi_k) dx dy dz \right. \\
& + \int_{\Omega} (b_1 \partial_x \psi_s \varphi_j + b_2 \psi_s \partial_y \varphi_j + b_3 \psi_s \partial_z \varphi_j) \psi_l \varphi_k dx dy dz \\
& \left. + \int_{\Omega} \sigma \psi_s \psi_l \varphi_j \varphi_k dx dy dz \right] \\
& = \int_{\Omega} f \psi_l \varphi_k dx dy dz \quad (1.9)
\end{aligned}$$

Osserviamo che il doppio indice " js ", in realtà scorre un vettore, la rimappatura in un solo indice può facilmente essere dedotta ottenendo che $[\mathbf{u}]_{js} = \mathbf{u}[(j-1)N + s]$.

La matrice generata ha dimensioni $(mN)^2$, tuttavia fissata la frequenza delle soluzioni e della funzione test è possibile identificare un blocco che corrisponde ad un problema monodimensionale. Se utilizziamo, in direzione x , gli elementi finiti di grado 1, il blocco risulta tridiagonale e, in questo caso, la matrice ha un numero di elementi non zero pari a $m^2(3N-2)$. Il pattern² di sparsità per un caso con $m=3$ e $N=14$ è riportato in figura 1.1.

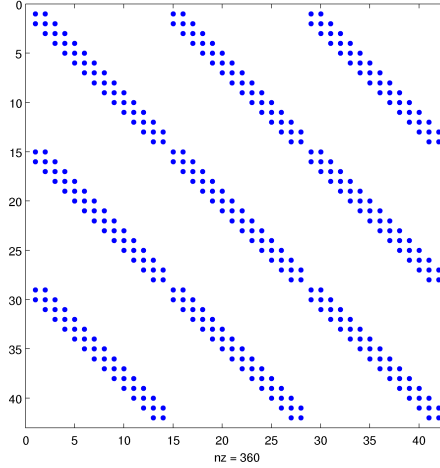


Figura 1.1: Pattern di sparsità per un caso con 14 elementi P1 e 3 modi.

²La matrice dei coefficienti è dunque sparsa ed inoltre il pattern è noto a priori, queste informazioni hanno permesso un assemblaggio più veloce in sede implementativa.

1.3 Educated Basis

Introduciamo con degli esempi l'algoritmo di ricerca delle basi ortonormali, senza entrare nel dettaglio della teoria;

1. **Costruzione di un problema ausiliario** che rispecchi la natura delle condizioni alle pareti del problema originale (devono essere omogenee) e passaggio ai relativi problemi agli autovalori.

Esempio - RRRR

Nel caso si abbiano condizioni di robin uguali sull'intera parete del vaso, dovremo considerare il seguente problema ausiliario:

$$\begin{cases} -\Delta u(y, z) = 0 & \text{in } \gamma_x \\ \mu \nabla u(y, z) \cdot \mathbf{n}_{\gamma_x} + \chi u(y, z) = 0 & \text{su } \Gamma_{vaso} \end{cases} \quad (1.10)$$

Si passi ora al problema agli autovalori associato al precedente sistema e ipotizzando la separazione di variabili per $u(y, z) = \varphi(y)\vartheta(z)$, si arrivano facilmente ad ottenere i seguenti sottoproblemi agli autovalori:

$$\begin{cases} -\varphi(y)'' = K_y \varphi(y) \\ \mu \varphi(y)' + \chi \varphi(y) = 0 & \text{per } y = L_y \\ -\mu \varphi(y)' + \chi \varphi(y) = 0 & \text{per } y = 0 \end{cases} \quad (1.11)$$

$$\begin{cases} -\vartheta(z)'' = K_z \vartheta(z) \\ \mu \vartheta(z)' + \chi \vartheta(z) = 0 & \text{per } z = L_z \\ -\mu \vartheta(z)' + \chi \vartheta(z) = 0 & \text{per } z = 0 \end{cases} \quad (1.12)$$

2. **Identificazione del tipo di soluzione** dei problemi agli autovalori associati.

Esempio - RRRR

Per i sottoproblemi ottenuti i generi di soluzione sono i seguenti:

$$\begin{aligned} \varphi(y) &= A \sin(\sqrt{K_y} y) + B \cos(\sqrt{K_y} y) \\ \vartheta(z) &= A \sin(\sqrt{K_z} z) + B \cos(\sqrt{K_z} z) \end{aligned} \quad (1.13)$$

3. **Ricerca degli autovalori di un sottoproblema** tramite risoluzione dell'equazione non lineare associata ad esso, ottenuta risolvendo le condizioni di bordo.

Esempio - RRRR

Nel caso trattato in esempio le equazioni che si ottengono sono le seguenti ($x = \sqrt{K_y}$ e $w = \sqrt{K_z}$):

$$\begin{aligned} f(x) &= 2\mu x + \tan(L_y x) \left(\chi - \frac{\mu^2 x^2}{\chi} \right) \\ f(w) &= 2\mu w + \tan(L_z w) \left(\chi - \frac{\mu^2 w^2}{\chi} \right) \end{aligned} \quad (1.14)$$

Osservazione

Nel caso di condizioni al bordo di Dirichlet il problema si semplifica. Infatti non occorre adottare l'algoritmo mostrato precedentemente, gli autovalori che si ottengono sono noti a priori e sono della forma:

$$\begin{aligned} K_y &= \left(\frac{\pi p}{L_y} \right)^2 & p &= 1, \dots, m_y \\ K_z &= \left(\frac{\pi q}{L_z} \right)^2 & q &= 1, \dots, m_z \end{aligned} \quad (1.15)$$

Dunque è nota la relazione $\lambda(K_y, K_z)$ a priori e risulta molto semplice ordinare in modo crescente gli autovalori definendo quindi m_y e m_z .

1.4 Ipotesi

Le ipotesi alla base di questo progetto sono le seguenti:

- Il dominio di calcolo è un parallelepipedo che si estende nell'ottante positivo.
- I coefficienti della forma sono assunti costanti.
- Viene risolto un problema ADR stazionario con condizioni di inflow di tipo Dirichlet e di outflow di tipo Neumann omogeneo.
- Condizioni sulle pareti omogenee.
- È possibile separare il problema lungo le direzioni trasversali, in due sotto problemi agli autovalori.

La forma del dominio considerato ci consente agilmente di applicare le tecniche di separazione di variabili e utilizzare la teoria delle basi educate. Più delicata risulterebbe la gestione di condizioni di bordo sulla parete del vaso, nel caso di sezione a forma generica. Nei capitoli successivi verranno accennate le difficoltà che presenta questa tematica. Anche nel caso di generalizzazione dei coefficienti della forma, viene presentata una soluzione possibile, tuttavia il codice è strutturato per l'utilizzo di coefficienti

costanti. In ogni caso consideriamo termini forzanti non costanti lungo il dominio. Per quanto il problema a sezione cilindrica potesse sembrare uno stretto parente del parallelepipedo così non è; renderemo chiare le principali differenze, insite nell'equazione risultante dalla separazione di variabili. Per quanto riguarda le condizioni di inflow, il codice permette di applicare condizioni di Dirichlet non omogenee, tale generalità non vale per la condizione di Neuamann all'outflow, ma l'eventuale estensione è triviale, dato che è sufficiente modificare opportunamente la forma bilineare.

Dalla teoria di HiMod sarà ormai chiaro che nella discretizzazione del dominio si fondono due concezioni molto diverse, da una parte gli Elementi Finiti lungo la fibra di supporto, dall'altra la base modale 2D, che ricorda molto i metodi spettrali. Dunque l'organizzazione delle classi è seguita naturalmente dalle necessità del metodo. Avevamo bisogno inizialmente di una classe che ci permettesse di maneggiare gli elementi dello spazio modale, ovvero la classe **ModalSpace**, ottenuto questo primo risultato è stata creata la classe che mette in comunicazione la fibra di supporto con le slices e risolve il problema, ovvero **HiModAssembler**. Il terzo soggetto principale di questo lavoro è **Basis1DAbstract**, ovvero la classe su cui si appoggia ModalSpace per costruire le basi modali basate sulla teoria delle basi educate. Nel prossimo capitolo analizzeremo nel dettaglio queste tre colonne portanti del codice.

Capitolo 2

Descrizione classi

2.1 Modalspace

Inizialmente ModalSpace è stata concepita per essere una classe base, dalla quale ereditasse ogni possibile scelta delle condizioni di bordo sulla parete del vaso. Facendo un rapido conto ci si accorge che comprendendo le condizioni di Dirichlet, Neumann e Robin su una sezione rettangolare arriviamo a 81 possibili combinazioni, una grande quantità di codice da scrivere, che comprende casistiche molto simili fra loro se non identiche. Questo è stato il primo motivo che ci ha portato a scorporare il trattamento delle condizioni di bordo dalla classe ModalSpace, per poi includerlo in modo ottimale in Basis1DAbstract (classe che si occupa della gestione delle basi educate). Un secondo punto a favore di questa scelta riguarda la valutazione e la lettura delle basi modali. Se avessimo scelto di adottare l'ereditarietà in ModalSpace, ogni eventuale figlia avrebbe avuto un tipo di base differente e accedervi tramite la classe base ogni qual volta fosse necessario, non risultava essere efficiente. Dunque la struttura di ModalSpace è composta dalle valutazioni delle basi modali su un'opportuna griglia e i generatori di basi (oltre ai metodi di calcolo necessari), questo design attribuisce maggiore generalità: ModalSpace è pronta ad utilizzare nuovi metodi in grado di generare una corretta base modale.

GraficoUMLstringatodimodalspaceebasis1dAbstract (2.1)

2.1.1 Costruzione e setting

ModalSpace conosce la geometria della sezione (L_y , L_z) e sicuramente deve conoscere il grado di accuratezza desiderato dall'utente, ovvero il numero di modi da utilizzare ($mtot$). Un altro punto fondamentale del costruttore generico è senz'altro la regola di quadratura da utilizzare sulla slice. Si noti che le basi utilizzate necessitano regole di quadratura di alto ordine e il

grado di esattezza è strettamente legato al numero di modi. Questo legame è evidente se si pensa che maggiore è il modo, maggiore sarà la frequenza della base modale e conseguentemente si avrà bisogno di una fitta successione di nodi di quadratura. Su una sezione quadrata una buona approssimazione dei nodi necessari su ciascun lato è \sqrt{mtot} . Il risultato non è valido nel caso di sezioni molto asimmetriche, infatti rettangoli molto allungati in una direzione avranno bisogno di più nodi lungo la direzione maggiore e meno sull'altra. Come esempio si osservi la seguente tabella dove sono riportati i check dei valori di normalità di una base, fissata la regola di quadratura al variare della dimensione L_y .

TABELLA (2.2)

Una volta creato l'oggetto `ModalSpace` bisogna eseguire alcuni set importanti. Per prima cosa dobbiamo impostare i generatori di base lungo le direzioni trasversali. Nel caso di basi educate questa operazione viene eseguita assieme all'imposizione delle condizioni di parete tramite i metodi pubblici:

- `void AddSliceBCY(const string left, const string right, const Real mu = 1, const Real Chi =1);`
- `void AddSliceBCZ(const string left, const string right, const Real mu = 1, const Real Chi =1)`

```
void ModalSpace::
AddSliceBCY (const string& left , const string& right , const
             Real& mu, const Real& chi)
{
    M_genbasisY = Basis1DFactory::instance().createObject(left+
        right);
    M_genbasisY->setL(M_Ly);
    M_genbasisY->setMu(M_mu);
    M_genbasisY->setChi(chi)

    return;
}
```

Nel caso si desideri aggiungere nuove tipologie di basi, queste dovranno essere ereditate dall'oggetto `Basis1DAbstract`, il quale conferisce una struttura generale al generatore di basi, osservando le ipotesi descritte nella sezione [1.4].

Aprire il discorso della factory, meglio farlo insieme (2.3)

Infine si conclude il seting della classe `ModalSpace` tramite la funzione membro pubblica `EvaluateBasis()`, che chiama le funzioni adibite a riempire le strutture dati, che mostriamo nella seguente sezione.

```
boost :: shared_ptr<ModalSpace> MB (new ModalSpace (Ly, Lz, mtot,
    quadY, quadZ));
MB ->AddSliceBCY("dir", "dir");
MB ->AddSliceBCZ("rob", "rob", 1., 3.);
MB ->EvaluateBasis();
```

Strutture dati

Diamo un breve descrizione delle strutture dati possedute dalla classe `ModalSpace`. Per prima cosa però, occupiamoci di un aspetto fondamentale. Le basi modali sono determinate sull'intervallo di riferimento, per non incorrere in errori fra dominio reale e riferimento, utilizzeremo la seguente notazione:

$$\begin{aligned} \hat{\varphi}_j(\hat{y}, \hat{z}) &= \hat{\eta}_j(\hat{y})\hat{\xi}_j(\hat{z}) \quad \hat{y} \in [0, 1] \quad \hat{z} \in [0, 1] \\ \int_0^1 \hat{\eta}_j^2 d\hat{y} &= 1 \quad \int_0^1 \hat{\xi}_j^2 d\hat{z} = 1 \end{aligned} \quad (2.4)$$

Dove $\hat{\varphi}_j$ è la base modale ortonormale sul dominio di riferimento, risultato del prodotto delle basi ottenute tramite i generatori di basi. Vediamo ora come gestire il passaggio dalle basi definite sul riferimento a quelle invece sul dominio reale. L'ortogonalità si conserva facilmente, ma lo stesso discorso non vale per la normalizzazione. Verifichiamo che un semplice cambio di coordinate non conserva la normalizzazione:

$$\begin{aligned} &\int_0^{L_y} \int_0^{L_z} \varphi_j(y, z)^2 dy dz \\ &= \int_0^{L_y} \eta_j(y)^2 dy \int_0^{L_z} \xi_j(z)^2 dz \\ &= \int_0^1 \eta_j(L_y \hat{y})^2 L_y d\hat{y} \int_0^1 \xi_j(L_z \hat{z})^2 L_z d\hat{z} \\ &= L_y L_z \int_0^1 \hat{\eta}_j(\hat{y})^2 d\hat{y} \int_0^1 \hat{\xi}_j(\hat{z})^2 d\hat{z} \quad \neq 1 \end{aligned} \quad (2.5)$$

Da questi semplici passaggi deduciamo che per essere mantenere la normalizzazione, la base che stiamo cercando avrà la seguente forma:

$$\varphi_j(y, z) = (L_y L_z)^{-\frac{1}{2}} \hat{\eta}_j(y L_y^{-1}) \hat{\xi}_j(z L_z^{-1}) \quad (2.6)$$

In conclusione, nei conti che verranno proposti si faccia sempre riferimento all'equazione (2.6).

Riconosciamo cinque strutture dati fondamentali per la classe `ModalSpace`:

- **EigenContainer M_eigenvalues**, contiene le sottofrequenze e gli indici corrispondenti, viene prodotta in fase di setting dello spazio modale tramite la funzione membro **EigensProvider()**, chiamata da **EvaluateBasis()**. Il tipo è un **vector<EigenMap>** dove:

```

struct EigenMap
{
    Real wp;    //subfrequency y
    Real wq;    //subfrequency z
    UInt p;
    UInt q;

    static EigenMap make_eigenmap(const Real& _wp, const Real&
        _wq, const UInt& _p, const UInt& _q)
    {
        EigenMap a;
        a.wp = _wp;
        a.wq = _wq;
        a.p = _p;
        a.q = _q;
        return a;
    }
};

```

L'ordinamento gerarchico degli autovalori e la corrispondenza delle sottofrequenze con i sottoindici è fondamentale, approfondiremo in seguito il metodo **EigensProvider()**.

- **MBMatrix_type M_phiy**, è un **vector<vector<Real> >** che raccoglie la valutazione di $\hat{\eta}_j(\hat{y}) \forall j$ e per ogni nodo di quadratura lungo $\hat{y} \in [0, 1]$.
- **MBMatrix_type M_phiz**, è un **vector<vector<Real> >** che raccoglie la valutazione di $\hat{\xi}_j(\hat{y}) \forall j$ e per ogni nodo di quadratura lungo $\hat{z} \in [0, 1]$.
- **MBMatrix_type M_dphiy**, è un **vector<vector<Real> >** che raccoglie la valutazione di $\frac{\partial \hat{\eta}_j}{\partial \hat{y}} \forall j$ e per ogni nodo di quadratura lungo $\hat{y} \in [0, 1]$.
- **MBMatrix_type M_dphiz**, è un **vector<vector<Real> >** che raccoglie la valutazione di $\frac{\partial \hat{\xi}_j}{\partial \hat{z}} \forall j$ e per ogni nodo di quadratura lungo $\hat{z} \in [0, 1]$.

2.1.2 Metodi di calcolo

Approfondiamo ora i metodi che si occupano di calcolare i coefficienti della matrice di sistema.

- `Real Compute_PhiPhi(const UInt& j, const UInt& k)`

$$\int_{\gamma_x} \varphi_j(y, z) \varphi_k(y, x) dydz$$
- `Real Compute_DyPhiPhi(const UInt& j, const UInt& k)`

$$\int_{\gamma_x} \partial_y \varphi_j(y, z) \varphi_k(y, x) dydz$$
- `Real Compute_DzPhiPhi(const UInt& j, const UInt& k)`

$$\int_{\gamma_x} \partial_z \varphi_j(y, z) \varphi_k(y, x) dydz$$
- `Real Compute_DyPhiDyPhi(const UInt& j, const UInt& k)`

$$\int_{\gamma_x} \partial_y \varphi_j(y, z) \partial_y \varphi_k(y, x) dydz$$
- `Real Compute_DzPhiDzPhi(const UInt& j, const UInt& k)`

$$\int_{\gamma_x} \partial_z \varphi_j(y, z) \partial_z \varphi_k(y, x) dydz$$
- `Real Compute_Phi(const UInt& k)`

$$\int_{\gamma_x} \varphi_k(y, x) dydz$$
- `vector<Real> FourierCoefficients (const function_Type& g) const,`
data una funzione indipendente da x questo metodo restituisce i coefficienti di Fourier (in numero pari ad M_mtot) rispetto alla base modale scelta.
- `Real Coeff_fk (const Real& x, const function_Type& f, const UInt& k) const,`
restituisce il k -esimo coefficiente di Fourier di una generica funzione 3D valutato nel punto x , rispetto alla base modale.

Date le premesse risulta ora semplice risolvere gli integrali scritti qui sopra, vediamo ad esempio che aspetto ha `Compute_PhiPhi()`:

```
Real ModalSpace::
Compute_PhiPhi(const UInt& j, const UInt& k) const
{
    Real coeff_y = 0.0;
    Real coeff_z = 0.0;
    UInt p_j = M_eigenvalues[j].p-1;
    UInt p_k = M_eigenvalues[k].p-1;
    UInt q_j = M_eigenvalues[j].q-1;
    UInt q_k = M_eigenvalues[k].q-1;

    Real normy = 1.0 / sqrt(M_Ly);
    Real normz = 1.0 / sqrt(M_Lz);

    for(UInt n = 0; n < M_quadruleY->nbQuadPt(); ++n)
    {
        coeff_y += M_phiy[p_j][n] * normy *
```

```

        M_phiy[p_k][n] * normy *
        M_Ly * M_quadruleY->weight(n);
    }

    for(UInt n = 0; n < M_quadruleZ->nbQuadPt(); ++n)
    {
        coeff_z += M_phiz[q_j][n] * normz *
                   M_phiz[q_k][n] * normz *
                   M_Lz * M_quadruleZ->weight(n);
    }

    return coeff_y * coeff_z;
}

```

Gli ultimi due metodi citati sono indispensabili ed il loro impiego sarà noto una volta che tratteremo la classe `HiModAssembler`.

2.1.3 EigensProvider()

Abbiamo deciso di dedicare una sezione solamente a questo metodo, poiché la ricerca degli autovalori occupa un ruolo fondamentale nella struttura del codice. Il metodo viene chiamato da `EvaluateBasis()` dunque dopo che sono stati settati i generatori di basi. La funzione deve preoccuparsi di definire la struttura dati `M_eigenvalues`, tuttavia il procedimento non è scontato. Per comprendere le difficoltà occorre ragionare sulla struttura del problema. Separando le variabili della slice 2D si sono ottenuti due problemi agli autovalori 1D (sezione [1.4]). Ognuno di questi genera una successione ordinata crescente di autovalori, determinata dalla risoluzione del problema agli autovalori, che nel caso di basi educate si traduce nella ricerca degli zeri di una data funzione non lineare. Definiamo la successione di autovalori in y con $\{K_y\}_p$ e quella in z con $\{K_z\}_q$. Le precedenti successioni definiscono univocamente la successione degli autovalori del problema di partenza 2D e sono in relazione con essa nel seguente modo:

$$\lambda_j = (K_y^p)^2 + (K_z^q)^2 \quad (2.7)$$

Anche $\{\lambda\}_j$ è una successione crescente di autovalori, ma il suo ordinamento, dato quello dei sottoautovalori, non è immediato. Due sono le difficoltà che si presentano:

1. Ogni sottoautovalore è il risultato di una ricerca di zeri di una funzione non lineare.
2. L'utente stabilisce il numero massimo di modi sul problema 2D e non sui sotto-problemi 1D.

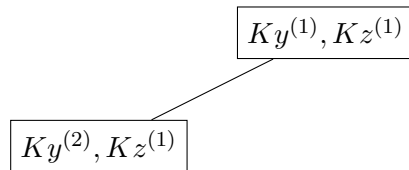
Le due problematiche sono strettamente legate, difatti non siamo interessati a cercare più sottoautovalori del necessario. Si poteva partire calcolando ad esempio 10 sottoautovalori in y e altrettanti in z , ordinare la successione 2D e procedere eventualmente nella ricerca. Questo metodo tuttavia presenta due difetti: è poco efficiente ed inoltre può cadere in errore. Infatti l'algoritmo si dovrebbe fermare una volta raggiunti un numero di autovalori 2D pari ad M_{mtot} , ma così facendo nessuno ci assicura che nel gruppo successivo di 10 autovalori non vi sia almeno uno minore dell'ultimo autovalore calcolato.

La soluzione è stata quella di procedere un passo alla volta, con l'accortezza di salvare i sotto-autovalori ancora non utilizzati. Per fare questo il metodo di ricerca degli zeri (`Next()` che approfondiremo nella sezione `Basis1DAbstract`) fornisce progressivamente uno zero alla volta. Infine abbiamo analizzato il seguente albero delle scelte:

```

Ky_1 = NextY();
Kz_1 = Nextz();
-> La prima coppia viene sempre inserita nella mappa
-> Cerco Ky_2 e Kz_2 nella mappa o nel registro
    if (trovati) -> li assegno a Ky_2 Kz_2;
    else      Ky_2 = NextY();
              Kz_2 = Nextz();
-> Inserisco nella mappa la coppia di sottoautovalori minore
-> Registro i sottoautovalori non inseriti

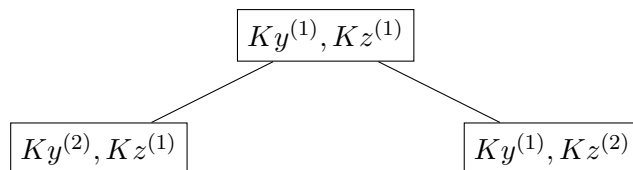
```



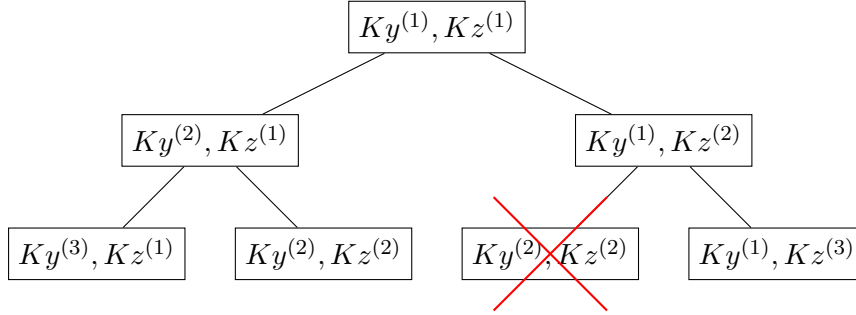
```

-> Cerco Ky_3 e Kz_2 nella mappa o nel registro
    if (trovati) -> li assegno a Ky_3 Kz_2;
    else      Ky_2 = NextY();
              Kz_2 = Nextz();
-> Inserisco nella mappa la coppia di sottoautovalori minore
    rispetto anche a quelli nel registro
-> Registro i sottoautovalori non inseriti

```



→ inserimento di copie evitato per unicità degli oggetti
nella struttura scelta
(Ky_{-2}, Kz_{-2}) già esistente non viene inserito nella mappa



2.2 Basis1dAbstract

Questa classe si occupa di definire dei generatori di base, ovvero gli oggetti puntati nella classe `ModalSpace` dai membri `M_genbasisY` e `M_genbasisZ`. Ereditando la struttura generale basata solamente sulle ipotesi [1.4], abbiamo implementato delle classi derivate seguendo la teoria delle Educated Basis (E.B.). La struttura polimorfica sembra bene adattarsi alle possibili evoluzioni future del problema, poiché diverse teorie sulla scelta delle basi sono tutt'ora in fase di sviluppo.

I metodi

La classe base è astratta, da essa derivano nove classi figlie, le quali corrispondono alle possibili combinazioni di condizioni di bordo omogenee di un problema 1D. Ogni classe figlia eredita pubblicamente da `Basis1DAbstract` ed implementa in modo proprio i seguenti metodi:

- `Real Next()`
- `void EvaluateBasis()`
- `Real EvalSinglePoint()`

I primi due metodi sono usati nella fase di set dell'oggetto `modalspace`, mentre l'ultimo è un'utilità della quale ne vedremo l'applicazione nella fase di export posseduta da `HiModAssembler`.

Il membro principale di ogni classe figlia è `M_ptrfunctor`. L'unica eccezione è il caso Dirichlet su entrambe i bordi, nell'osservazione in [1.3] abbiamo sottolineato la trivialità di questo caso, che non necessita la definizione del funtore. L'oggetto puntato da questo `shared_pointer` è un

EducatedBasisFunctor. Abbiamo scelto di costruire un sistema di classe base e derivate ausiliario, simile strutturalmente a **Basis1DAbstract**: la classe base è costituita da **EducatedBasisFunctorAbastract** e vi è una classe figlia per ogni possibile combinazione di condizioni di bordo di un problema 1D (dunque nove come le classi figlie di **Basis1DAbstract**), per ognuna di esse è implementato solamente l'operatore $()$. Ogni funzione ricavata dalla risoluzione dei problemi agli autovalori, è contenuta in questa gerarchia di classi ausiliarie. Il costruttore è comune a tutte le classi figlie, ed è quindi definito solamente nella classe base, esso si occupa di settare in maniera corretta i parametri riferiti alle condizioni di bordo applicate (μ e χ).

I restanti membri hanno l'utilità di memorizzare quale autovalore è stato calcolato per ultimo, in questo modo il metodo **Next()** è in grado di procedere al calcolo progressivo di tutti gli autovalori desiderati, senza tornare sui suoi passi. Per comprendere meglio la distribuzione degli autovalori, mostriamo nella seguente figura la forma della funzione non lineare nel caso Robin-Robin.

FIGURA (2.8)

Abbiamo visto che **ModalSpace** possiede il metodo **EvaluateBasis()**, in realtà questa funzione chiama separatamente i metodi omonimi, di proprietà dei due generatori di basi, poiché sono loro a conoscere la forma delle funzioni di base. L'implementazione di **EvaluateBasis()** è semplice, l'obiettivo è il completamento di una delle strutture dati matriciali possedute da **ModalSpace**. In particolare il generatore **M_genbasisY** si occuperà di **M_phiy** e **M_dphey**, mentre **M_genbasisZ** completerà le restanti strutture. Facendo riferimento alla forma generica delle basi modali (1.13), è chiaro che il compito di **EvaluateBasis()** è di calcolare i coefficienti A e B in modo tale che le basi rispettino le condizioni di bordo e risultino normali (ricordiamo che l'ortogonalità è garantita dalla teoria).

La registrazione delle valutazioni della base modale nei nodi di quadratura è un'operazione necessaria al fine di velocizzare le operazioni di integrazione. Tuttavia come procediamo se siamo interessati a valori delle basi su una griglia più fitta di quella di quadratura? Per risolvere questo problema è stato scritto il metodo **EvalSinglePoint()** che dato un sotto autovalore e la coordinata, restituisce la valutazione della base modale in quel dato punto. Questa funzione si è resa necessaria dal momento che in fase di export desideravamo interpolare la funzione su una griglia con più nodi rispetto alla quadratura.

2.3 HiModAssembler

In questa classe viene gestita la fase di assemblaggio del problema e quella di export. Abbiamo cercato di seguire la linea della classe FESpace, infatti `HiModAssembler` è templetizzata sugli stessi argomenti. Oltre a mantenere una struttura di base coerente con lo spazio agli elementi finiti, la scelta di templetizzare le matrici e i vettori, manipolati da tale classe, prevede una possibile implementazione futura di strutture algebriche più adatte al metodo di riduzione gerarchica. Ricordiamo infatti che la struttura della matrice di sistema è ben nota (Fig.1.1), formata da un numero di blocchi pari ad M_{mtot}^2 , ognuno dei quali di dimensione pari al quadrato dei gradi di libertà FEM spesi lungo la fibra di supporto.

Viene templetizzata anche la mesh, ma questo parametro si riferisce discretizzazione della fibra di supporto, dunque siamo vincolati a trattarlo in questo modo. Momentaneamente il codice funziona solamente con elementi del tipo `EpetraStructured`, tuttavia occorrerebbe ripensare attentamente questa parte, soprattutto in visione di una possibile parallelizzazione. Un codice HiMod ottimizzato dovrà possedere delle proprie strutture, adeguate alla conformazione della matrice di sistema.

```
template<typename mesh_type,typename matrix_type,typename
        vector_type>
class HiModAssembler
{
    ...
};
```

I membri

La classe `HiModAssembler` possiede tre membri privati:

- `modalbasis_ptrType M_modalbasis`
- `fespace_ptrType M_fespace`
- `etfespace_ptrType M_etfespace`

Nota la teoria era chiaro che la classe dovesse possedere lo spazio modale 2D e lo spazio elementi finiti 1D. Il restante oggetto esiste per una scelta di programmazione. Infatti abbiamo deciso di ricorrere all'utilizzo del pacchetto `ETA`, piuttosto che del `GeneralAssembler`. Due sono le motivazioni che hanno condotto a questa scelta:

1. Semplicità di scrittura della forma variazionale.

2. Possibilità di scrivere più parti di forma variazionale.

Entrambe le motivazioni sono legate in realtà a possibili future estensioni del codice. Se pensiamo al problema generalizzato ad una qualunque sezione 2D, siamo costretti ad adottare una mappa che legghi lo spazio reale e lo spazio di riferimento (dove viene risolto il problema). I conti mostrano che alla forma variazionale si aggiungono numerosi casi non gestiti dal pacchetto **GeneralAssembler**.

L'utilizzo del modulo **ETA** è tuttavia nascosto all'utente, il costruttore si occupa di inizializzare **M.etfespace** ricavando le informazioni necessarie da **M.fespace**.

```
template<typename mesh_type,typename matrix_type,typename
vector_type>
HiModAssembler<mesh_type, matrix_type, vector_type>::
HiModAssembler( const fespace_ptrType& fespace,
                const modalbasis_ptrType& modalbasis,
                commPtr_Type& Comm):
    M.Modalbasis ( modalbasis ),
    M.etfespace ( new etfespace_type ( fespace->mesh(),
                                     &(fespace->refFE()),
                                     &(fespace->fe().geoMap()),
                                     Comm))
    M.fespace ( fespace)
{}
```

2.3.1 I metodi

Proponiamo di seguito una descrizione dei vari metodi disponibili nella classe **HiModAssembler**. La classe è ampia, tuttavia comprende tre sezioni ben distinte: assemblaggio, analisi e export.

Assemblaggio

```
void AddADRProblem ( const matrix_ptrType& systemMatrix,
                    const Real& mu,
                    const TreDvector_type& beta,
                    const Real& sigma)
```

Si occupa dell'assemblaggio della matrice di sistema. I coefficienti sono considerati costanti (ipotesi [1.4]), l'estensione a coefficienti non costanti comporta cambiamenti anche nella classe **ModalSpace**, tali modifiche non sono state apportate perchè esuli dai nostri obiettivi progettuali. Si è ragionato

su una possibile implementazione che segue la linea generale proposta negli articoli (citazione perotto). Le fasi di assemblaggio sono semplici: la matrice di sistema viene percorsa a blocchi e per ognuno di essi il computo dei valori avviene tramite il metodo `integrate` del pacchetto ETA. Difatti ogni blocco corrisponde al problema 1D che accoppia la frequenza j e k .

```
void interpolate ( const function_Type& f,
                  const vector_ptrType& f_interpolated )
```

Data una generica funzione spaziale, questo metodo si occupa di calcolare, per ogni punto della griglia 1D, tutte le componenti di Fourier rispetto alle basi modali. Il risultato viene salvato nel vettore strutturato passato negli argomenti. È in questo punto che ricopre un ruolo fondamentale il metodo `Coeff_fk()` posseduto da `ModalSpace`. In pratica, come sottolineato dalla firma stessa del metodo, stiamo interpolando una generica funzione sullo spazio HiMod. Le dimensioni del vettore che contiene i coefficienti di interpolazione sono pari ovviamente a `Mtot·DOFfem`.

```
void Addrhs ( const vector_ptrType& rhs,
              const vector_ptrType& f_interpolated );
```

Definito il metodo `interpolate` risulta semplice assemblare il termine noto. Ottenuta l'interpolazione della forzante l'approccio non è differente da `AdADRProblem()`: il vettore `rhs` possiede `Mmtot` blocchi di dimensione ciascuno pari ai gradi di libertà FEM, si scorrono tutti i blocchi e poichè ognuno di essi è legato al problema 1D riferito alla j -esima frequenza, il metodo `integrate()` computa gli opportuni coefficienti.

Mostriamo in breve le operazioni eseguite nel main per definire ed assemblare il termine noto:

```
boost::shared_ptr<vector_Type> rhs
    (new vector_Type (Map, Repeated));
*rhs *= 0.0;
rhs -> setBlockStructure (block_row);

boost::shared_ptr<vector_Type> f_interpolated
    (new vector_Type (Map, Repeated));

HM.interpolate ( f, f_interpolated );
HM.Addrhs (rhs, f_interpolated );
```

```

void AddDirichletBC_In (   const matrix_ptrType& systemMatrix,
                           const vector_ptrType& rhs,
                           const function_Type& g)

```

L'applicazione delle condizioni di inflow ed outflow sono un aspetto secondario di questo lavoro, tuttavia non possono certo essere esenti da un'adeguata trattazione. Per quanto riguarda le condizioni naturali del problema è sufficiente intervenire nella forma variazionale, nel caso invece di condizioni essenziali quali quelle di Dirichlet, abbiamo deciso di intervenire con una penalizzazione algebrica, molto simile al trattamento delle BC fatto da FreeFem++. Per capire dove intervenire dobbiamo rifarci all'interpretazione data nei cenni teorici, ricordiamo infatti che la matrice di sistema del metodo HiMod è costituita da M_{mtot}^2 problemi 1D correlati fra loro. Dunque se ogni sottoblocco rappresenta la matrice di sistema di un problema ADR agli elementi finiti 1D, è chiaro che sarà sufficiente intervenire sul primo elemento (nel caso di Dirichlet all'inflow). Analogamente l'intervento sul termine noto, viene eseguito sostituendo al primo elemento di ogni sottoblocco, il coefficiente dell'interpolazione del dato in ingresso, moltiplicato per il medesimo coefficiente di penalizzazione usato nella matrice di sistema.

Nel caso dell'interpolazione del dato in ingresso, si tratta di una situazione più semplice dell'interpolazione 3D, dunque è stato sviluppato il metodo `FourierCoefficients()` contenuto in `ModalSpace`.

```

template<typename mesh_type, typename matrix_type, typename
vector_type>
void HiModAssembler<mesh_type, matrix_type, vector_type>::
AddDirichletBC_In (   const matrix_ptrType& systemMatrix,
                       const vector_ptrType& rhs,
                       const function_Type& g)
{
    vector<Real> FCoefficients_g;
    FCoefficients_g = M_modalbasis->FourierCoefficients (g);
    UInt dof = M_etfespace->dof().numTotalDof();
    for( UInt j(0); j<M_modalbasis->mtot(); ++j)
    {
        systemMatrix->setCoefficient(j*dof, j*dof, 1e+30);
        rhs->setCoefficient(j*dof, 1e+30*FCoefficients_g[j]);
    }

    return;
}

```

Analisi

I seguenti metodi si occupano di manipolare le informazioni contenute nel vettore soluzione o in un generico vettore che raccoglie i coefficienti relativi all'interpolazione nello spazio HiMod. Il nostro primo interesse è poter ricostruire i valori della soluzione nei punti della griglia 3D costituita dai nodi di quadratura e i nodi FEM. Il vettore ottenuto sarà poi utilizzato per eventuali operazioni o analisi quali computo della norma L2. Per eseguire confronti con generiche funzioni abbiamo implementato anche una specializzazione nel caso l'argomento di ingresso risulti essere una generica funzione spaziale.

Ai fini di proporre confronti e grafici di convergenza è stata implementato il computo della norma L2, spazio principale in cui la teoria ha dimostrato stime di convergenza.

```
vector_type evaluateBase3DGrid (const vector_type& fun)

vector_type evaluateBase3DGrid (const function_Type& fun)

Real normL2 (const vector_type& fun)
```

La particolarità dei metodi appena presentati è il fatto che non viene valutata nessuna funzione, si tratta solamente di recuperare i valori memorizzati nelle strutture dati di `M_modalspace`. Tuttavia questo vantaggio è preservato fin tanto ci si accontenta di valutare la funzione nei punti della griglia di quadratura. Nel caso di una valutazione più fitta o di un punto arbitrario, occorre risalire alla forma originale delle basi modali. Questo è concesso grazie al metodo `EvalSinglePoint()`, posseduto da ogni generatore di basi il quale conosce, tramite la classe ausiliaria `EducatedBasisFunctor`, la forma originale della base modale. Ovviamente tale approccio si poteva adottare anche per i precedenti metodi, ma l'utilizzo avrebbe comportato la valutazione di una funzione, posseduta da un oggetto posto nelle foglie di una struttura polimorfica. Il risultato finale sarebbe stato un costo computazionale notevolmente maggiore.

```
Real evaluateHiModFunc(const vector_ptrType& fun, const Real&
x, const Real& y, const Real& z)
```

Export

Particolare attenzione meritano i metodi dedicati all'export. Ci siamo occupati di esportare in formato VTK considerando due situazioni differenti:

1. Griglia strutturata.

2. Griglia non strutturata.

```
void ExporterStructuredVTK (  const UInt& nx,
                             const UInt& ny,
                             const UInt& nz,
                             const vector_ptrType& fun ,
                             const GetPot& dfile ,
                             string prefix)
```

Nel primo caso si cerca di lavorare sulla struttura ordinata del problema
Ricordiamo che:

$$u(x, y, z) = \sum_{j,s}^{m,n} u_{js} \psi_s(x) \varphi_j(y, z) = \sum_{j,s}^{m,n} u_{js} \psi_s(x) \eta_p(y) \xi_q(z) \quad (2.9)$$

Vi sono due modi di procedere, o fissiamo il modo e calcoliamo tutti i contributi o lo facciamo per la funzione FEM. Noi abbiamo scelto di farlo sui modi, quindi il ciclo più esterno è sui modi. **Sarebbe interessante provare a fare l'altro modo per vedere se ci si mette di meno**

```
void ExporterGeneralVTK (  const export_mesh_Type& mesh,
                           const vector_ptrType& fun ,
                           const GetPot& dfile ,
                           string prefix)
```

Capitolo 3

Risultati

- Introduzione teorica
- Osservazione sull'influenza dell'errore fem (particolare nel caso RRRR sarebbe da verificare)
- Notare la superconvergenza (come da teoria?)

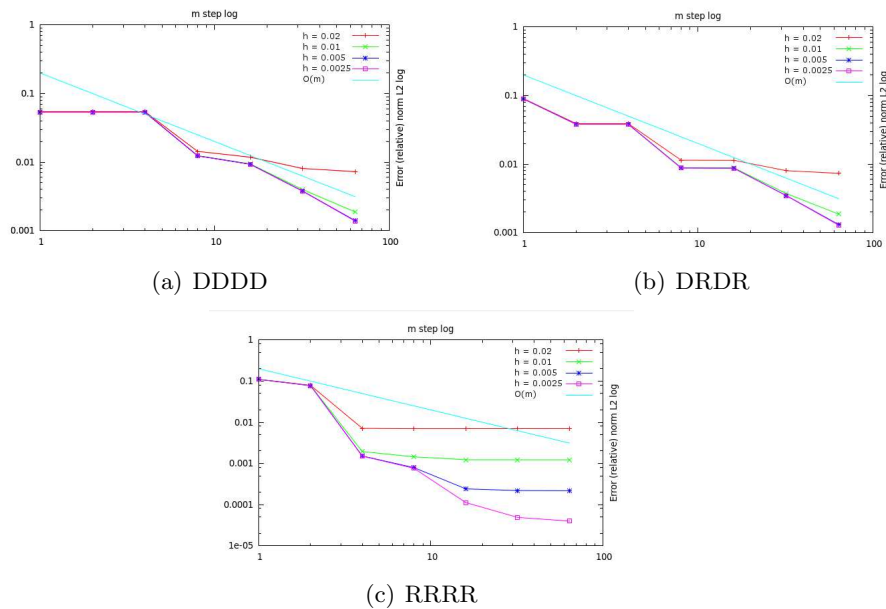


Figura 3.1: Convergenze casi test

- Confronto gradi di libertà spesi e errore norma L2
- Tempo di assemblaggio problema

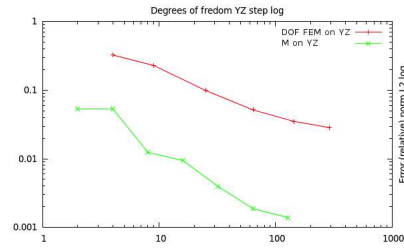


Figura 3.2: Confronto DOF

- Presentazione del caso test
- Osservazione sull'aggiunta progressiva di modi (sempre più frequenze vengono considerate e l'approssimazione cresce)
- Commento sui risultati 3D

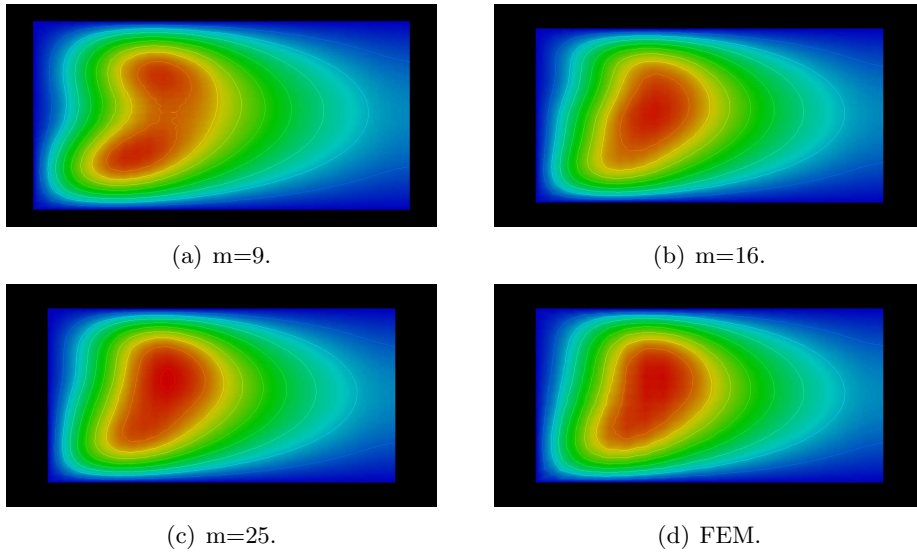
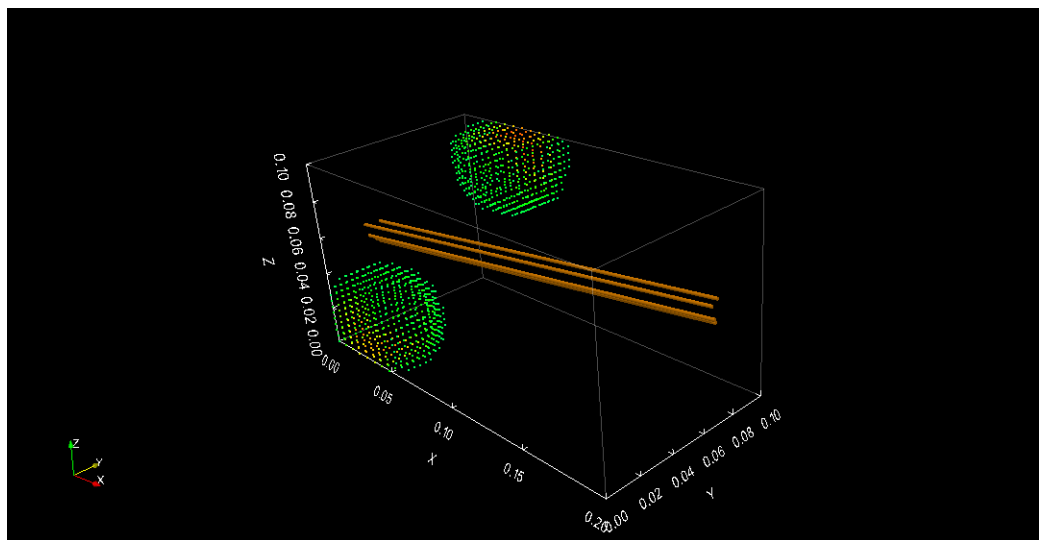
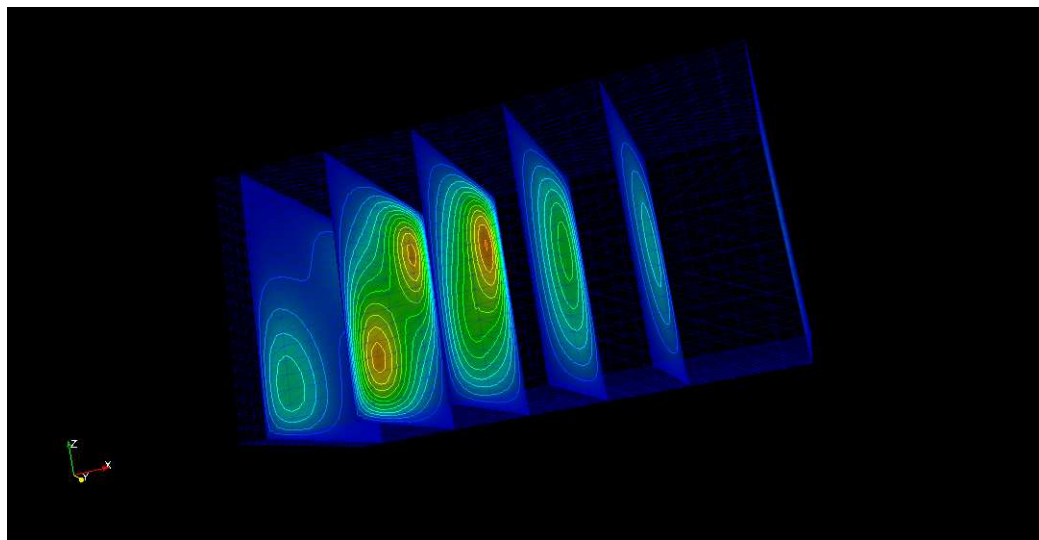


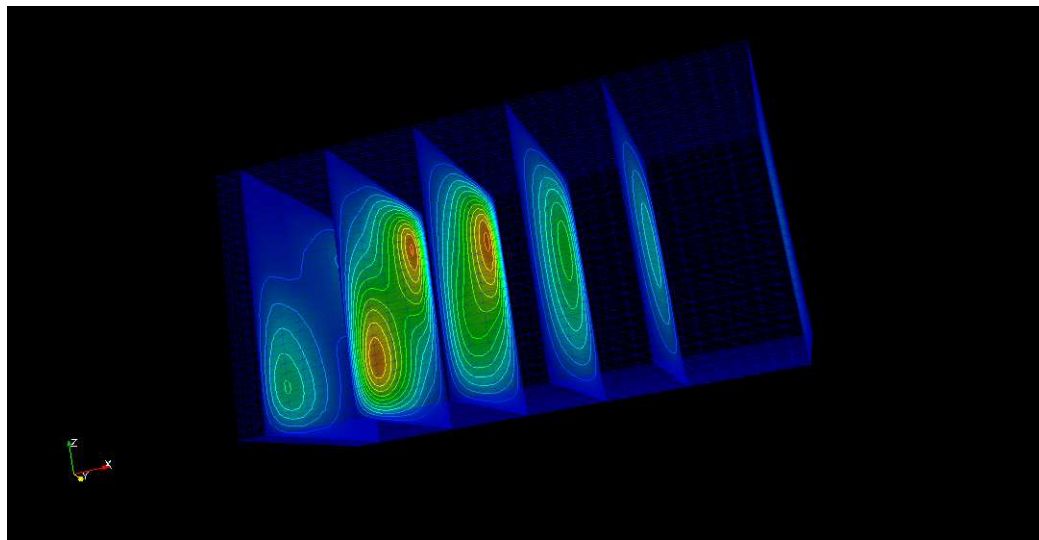
Figura 3.3: Caso test DDDD sezione XZ



(a) Forzante e trasporto.



(b) Soluzione FEM.



(c) Soluzione HiMod $m=50$.