Laurea magistrale
in ingegneria matematica

Progetto del corso di
Programmazione Avanzata per il Calcolo Scientifico.


# Implementazione in LifeV del metodo di Riduzione Gerarchica di Modello

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

Anno Accademico 2012–2013

# Indice

# Capitolo 1

# Introduzione

In molti problemi di interesse ingegneristico i fenomeni in esame presentano direzioni preferenziali. Ad esempio è possibile incontrare problemi simili in emodinamica dove la direzione del flusso è dominante rispetto alla dinamica che avviene in direzione trasversale. L'obbiettivo di questo progetto è l'implementazione in `LifeV` di un risolutore per un problema di diffusione trasporto reazione ( Advection Diffusion Reaction - ADR) 3D, basato sulla tecnica di Riduzione Gerarchica di Modello (Hierarchical Model Reducation - HiMod).

HiMod si propone di sfruttare l'informazione sulla presenza di una direzione dominante per ridurre il costo computazionale della risoluzione del problema convertendo un problema tridimensionale in diversi problemi monodimensionali arricchiti. La riduzione è resa possibile da uno sviluppo della soluzione in serie di Fourier generalizzate nella direzione trasversale che consente di approssimare soltanto i primi coefficienti di Fourier della soluzione.

È chiaro che il metodo non potrà essere competitivo rispetto al metodo degli elementi finiti nel caso di problemi senza dinamiche dominanti, tuttavia, dove la dinamica trasversale è semplice, consente di ottenere una buona approssimazione del fenomeno, superiore rispetto a un modello ridotto monodimensionale, ma senza i costi di una risoluzione 3D.

Approfondimenti riguardo alla Riduzione Gerarchica di Modello si possono trovare in [EPV08],[PEV09],[PZ12] e in [Zil10] dove il metodo è stato applicato in un contesto bidimensionale e parzialmente sviluppato dal punto di vista teorico nel caso tridimensionale.

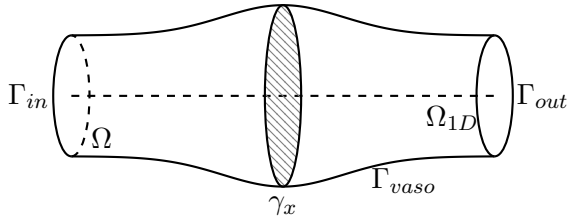
Nel progetto ci siamo focalizzati sull'implementazione del metodo in un contesto tridimensionale con una geometria semplice. In particolare abbiamo implementato le basi istruite: una particolare scelta della base di Fourier per la direzione trasversale in grade di incorporare le condizioni al bordo.

## 1.1 Nozioni base

In questa sezione presenteremo le basi teoriche del metodo, applicandolo a un problema di diffusione trasporto e reazione.

Consideriamo il seguente problema in un generico dominio $\Omega$, visto che il metodo è stato pensato soprattutto per applicazioni in emodinamica consideriamo un dominio di forma tubolare

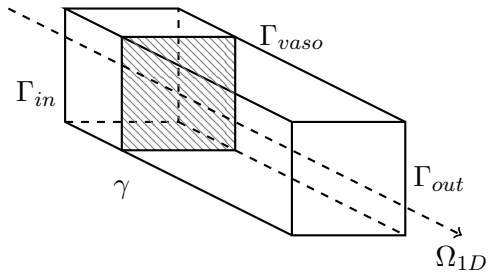
$$\begin{cases} -\mu\Delta u + \mathbf{b}\cdot\nabla u + \sigma u = f & \text{in } \Omega \\ u = u_{in} & \text{su } \Gamma_{in} \\ \frac{\partial u}{\partial \mathbf{n}} = 0 & \text{su } \Gamma_{out} \\ u = 0 & \text{su } \Gamma_{vaso} \end{cases} \tag{1.1}$$




dove i coefficienti che compaiono nell'equazione sono tali per cui il problema variazionale associato sia ben posto in $V = H^1_{\Gamma_{IN}\cup\Gamma_{vaso}}(\Omega)$. Immaginiamo di suddividere il dominio $\Omega$ in slice poste trasversalmente alla direzione longitudinale.

$$\Omega = \bigcup_{x\in\Omega_{1D}} \gamma_x \tag{1.2}$$

Ogni slice verrà indicata con $\gamma_x$. Lungo $\gamma_x$ verranno utilizzate funzioni spaziali differenti rispetto a quelle utilizzate lungo $\Omega_{1D}$. Nel caso del tubo a sezione rettangolare, sul quale il codice si focalizzerà, $\Omega$ si riduce a $(0, L_x)\times\gamma$ dove $\gamma_x = \gamma = (0, L_y)\times(0, L_z) \quad \forall x \in (0, L_x)$



Per gestire un dominio di forma generica è necessario utilizzare una mappa con la quale ricondursi ad un dominio di riferimento. La teoria delle basi istruite, tuttavia, non è, ora, in grado di coprire il caso della mappa[1].

Introduciamo alcuni spazi funzionali utili per ambientare correttamente il metodo. Su $\Omega_{1D}$ utilizziamo lo spazio $V_{1D} = H^1_{\Gamma_{in}}(\Omega_{1D})$, mentre sulla fibra trasversale $\gamma$ introduciamo le basi modali $\{\varphi_k\}_{k=1}^{\infty}$ ortonormali rispetto al prodotto scalare di $L^2(\gamma)$. Quest'ultime definiscono su $\gamma$ lo spazio funzionale $V_\gamma := span\,(\{\varphi_k\}_{k=1}^{\infty})$. È possibile dimostrare che, chiuso rispetto ad una opportuna norma, lo spazio $V_{1D} \otimes V_\gamma$ è isometricamente isomorfo a $V$.

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

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

Questo è l'ambiente funzionale in cui opera il metodo HiMod. L'ortogonalità in $L^2(\gamma)$ implica che i coefficienti $\tilde{v}_k$ che compaiono nella (1.3) siano il risultato del seguente prodotto scalare

$$\tilde{v}_k(x) = \int_\gamma \varphi_k(y,z) v_m(x,y,z)\, dydz \quad \forall k = 1 \dots m$$

osserviamo come questi rappresentino, puntualmente, i coefficienti di Fourier della soluzione esatta rispetto alla base utilizzata sulla fibra trasversale.

La convergenza di una soluzione $u_m$ tale che soddisfi il problema (1.1), nella sua forma variazionale posta sul sottospazio $V_m$, discende dalle seguenti proprietà:

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

Quest'approccio può essere esteso al caso di condizioni sulla parete laterale diverse dalle condizioni di Dirichlet omogenee. In questo progetto abbiamo implementato l'approccio delle basi istruite, ma altre scelte sono possibili.

[1] In fase di implementazione abbiamo cominciato ad inserire le funzionalità relative alla mappa, ma ci siamo limitati ad aprire un branch sul repository che, pur essendo praticamente completato, abbiamo dovuto abbandonare essendo incompatibile con le basi istruite. Con un altro tipo di base, ad esempio i polinomi di Legendre, è possibile recuperare il branch e completare l'implementazione.

## 1.2 Forma matriciale

Per ogni $m \in \mathbb{N}$ consideriamo il seguente problema ridotto
*Trovare* $u_m \in H^1(\Omega_{1D}) \otimes V_\gamma^m$ *tale che* $u_m|_{\Gamma_{in}}$ *sia uguale alla proiezione del dato di Dirichlet su* $V_\gamma^m$ *e valga*

$$\int_\Omega (\mu \nabla u_m \nabla v_m + \mathbf{b} \cdot \nabla u_m v_m + \sigma u_m v_m)\, d\Omega \quad \forall v_m \in V_m = \int_\Omega f v \, d\Omega \quad (1.4)$$

Utilizziamo l'espansione di $u_m(x, y, z)$ rispetto alla base di Fourier

$$u_m(x, y, z) = \sum_{j=k}^{m} \tilde{u}_j(x)\varphi_j(y, z), \quad \tilde{u}_j(x) = \int_\gamma u_m(x, y, z)\varphi_j(y, z)\, dydz$$

consideriamo inoltre funzioni test della forma

$$v_m = \vartheta(x)\varphi_k(y, z), \quad \vartheta(x) \in V_{1D} \text{ e } k = 1, ...m.$$

Il problema assume la seguente forma:

$$\begin{aligned} &\sum_{j=1}^{m} \int_\Omega \mu \nabla(\tilde{u}_j(x)\varphi_j(y, z))\nabla(\vartheta(x)\varphi_k(y, z))\, dxdydz \\ &+ \int_\Omega (\mathbf{b}\nabla(\tilde{u}_j(x)\varphi_j(y, z)) + \sigma\tilde{u}_j(x))\vartheta(x)\varphi_k(y, z)\, dxdydz \\ &= \int_\Omega f\vartheta(x)\varphi_k(y, z)\, dxdydz \end{aligned} \quad (1.5)$$

Svolgendo l'operatore gradiente si ottiene:

$$\begin{aligned} &\sum_{j=1}^{m} \int_\Omega \mu(\partial_x\tilde{u}_j\partial_x\vartheta\varphi_j\varphi_k + \tilde{u}_j\vartheta\partial_y\varphi_j\partial_y\varphi_k + \tilde{u}_j\vartheta\partial_z\varphi_j\partial_z\varphi_k)\, dxdydz \\ &+ \int_\Omega (b_1\partial_x\tilde{u}_j\varphi_j + b_2\tilde{u}_j\partial_y\varphi_j + b_3\tilde{u}_j\partial_z\varphi_j)\vartheta\varphi_k\, dxdydz \\ &+ \int_\Omega \sigma\tilde{u}_j\vartheta\varphi_j\varphi_k\, dxdydz \\ &= \int_\Omega f\vartheta\varphi_k\, dxdydz \end{aligned} \quad (1.6)$$

Per semplicità consideriamo una partizione $\tau_h$ uniforme lungo la fibra di supporto 1D. Sia $N$ il numero di vertici lungo $\Omega_{1D}$. Il passo della partizione è dunque $h = |\Omega_{1D}|/(N-1)$.

Introduciamo lo spazio agli elementi finiti lungo $\Omega_{1D}$

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

Per semplicità supponiamo di utilizzare elementi finiti di grado uno, ma la trattazione teorica è del tutto equivalente nel caso si volessero usare polinomi di grado più alto. Una volta introdotta la discretizzazione lungo la direzione dominante del fenomeno è possibile esprimere i coefficienti di Fuorier nel seguente modo

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

Abbiamo quindi discretizzato completamente il problema. Tramite l'espansione modale siamo stati in grado di ridurre il problema da 3D a $m$ problemi 1D accoppiati che ora abbiamo discretizzato con il metodo degli elementi finiti.

Otteniamo dunque la formulazione matriciale del problema ovvero
*Trovare* $\mathbf{u} \in \mathbb{R}^{N*m}$ *tale che*

$$\begin{aligned}
&\sum_{j=1}^{m}\sum_{s=1}^{N} u_{js}\Bigg[\int_\Omega \mu(\partial_x\psi_s\partial_x\psi_l\varphi_j\varphi_k + \psi_s\psi_l\partial_y\varphi_j\partial_y\varphi_k + \psi_s\psi_l\partial_z\varphi_j\partial_z\varphi_k)\,dxdydz \\
&+ \int_\Omega (b_1\partial_x\psi_s\varphi_j + b_2\psi_s\partial_y\varphi_j + b_3\psi_s\partial_z\varphi_j)\psi_l\varphi_k\,dxdydz \\
&+ \int_\Omega \sigma\psi_s\psi_l\varphi_j\varphi_k\,dxdydz\Bigg] \\
&= \int_\Omega f\psi_l\varphi_k\,dxdydz \quad \forall\psi_l \quad l = 1\dots N \quad \forall\varphi_k \quad k = 1\dots m
\end{aligned} \tag{1.9}$$

Per chiarezza abbiamo utilizzato un doppio indice $''js''$. Esso scorre in realtà un vettore, ma è possibile usare un solo indice che si lega a $''js''$ nel seguente modo

$$\mathbf{u}_{js} = \mathbf{u}[i] = \mathbf{u}[(j-1)N + s].$$

La matrice generata ha dimensioni $(mN)^2$, tuttavia fissata la frequenza della soluziona e della funzione test, ossia fissando l'indice che scorre la base modale, è possibile identificare un blocco che corrisponde ad un problema monodimensionale. Se utilizziamo gli elementi finiti di grado uno, il blocco è tridiagonale e la matrice ha un numero di elementi non zero pari a $m^2(3N-2)$. Il pattern di sparsità per un caso con m=3 e N=14 è riportato in figura 1.1. La matrice dei coefficienti è sparsa con un pattern noto a priori, ciò ha permesso un assemblaggio più veloce in fase implementativa. È anche possibile riordinare la matrice in modo differente. Utilizzando una struttura a blocchi associata ai gradi di libertà degli elementi finiti e non alla base modale. In questo modo avremmo una matrice tridiagonale a blocchi e ogni blocco sarebbe una matrice $m \times m$ piena. Con questa riordinamento della matrice la struttura è dunque quella di un problema 1D dove invece che un solo grado di libertà associato al nodo ne sono associati $m$. Abbiamo

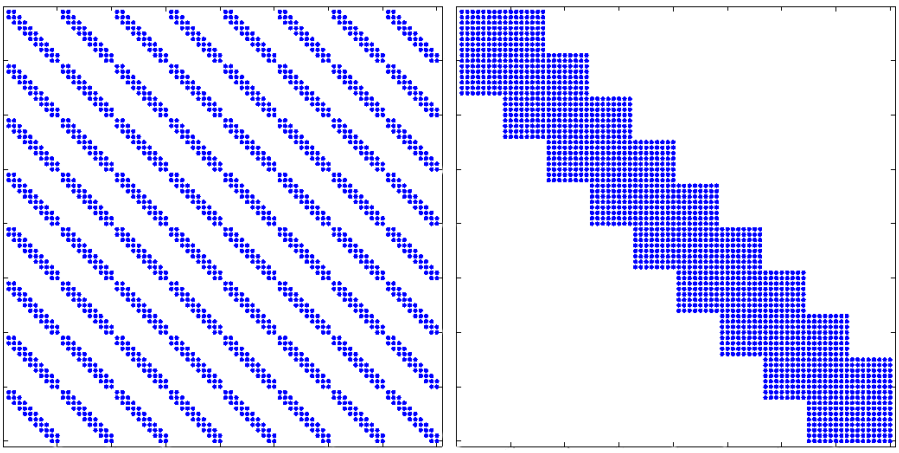


**Figura 1.1:** Pattern di sparsità per un caso con 9 elementi P1 e 8 modi.
A sinistra il pattern che è stato poi utilizzato, a destra l'alternativa.

implementato la prima scelta, che è quella suggerita in [**?** ], anche per poter sfruttare le procedure di assemblaggio per gli elementi finiti 1D già presenti in LifeV.

## 1.3 Basi istruite

Le basi istruite sono state chiamate in questo modo perchè sono basi in grado di leggere la natura delle condizioni al bordo e di incorporarla all'interno della base stessa. Esse conoscono parte del problema in esame. Utilizzare le basi istruite equivale a imporre in maniera essenziale anche le condizioni al bordo naturali, come quelle di tipo Robin. Il metodo delle basi istruite si basa sulla risoluzione di un problema agli autovalori ausiliario posto sulla fibra trasversale $\gamma$.[2] Introduciamo con un esempio l'algoritmo di costruzione delle basi istruite

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 - Robin BC

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

$$\begin{cases} -\Delta u(y,z) = 0 & \text{in } \gamma \\ \mu \nabla u(y,z) \cdot \mathbf{n} + \chi u(y,z) = 0 & \text{su } \partial\gamma \end{cases} \tag{1.10}$$

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

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

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

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

Esempio - Robin BC

Per i sottoproblemi ottenuti i generi di soluzione sono i seguenti

$$\begin{aligned} \varphi(y) &= A_y sin(\sqrt{K_y}y) + B_y cos(\sqrt{K_y}y) \\ \vartheta(z) &= A_z sin(\sqrt{K_z}z) + B_z cos(\sqrt{K_z}z) \end{aligned} \tag{1.13}$$

[2] Nell'implementazione abbiamo risolto il problema riportandolo sul quadrato di riferimento $(0,1) \times (0,1)$.

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

Esempio - Robin BC

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

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

Per giustificare l'algoritmo dal punto di vista teorico riportiamo solamente un teorema che analizza un generico problema agli autovalori. Per adattarlo al nostro caso è sufficiente scegliere $H = L^2(\gamma)$ e $V = H^1_{\partial\gamma_o}(\gamma)$, dove $\partial\gamma_0$ è il sottoinsieme di $\partial\gamma$ dove sono imposte le condizioni di Dirichlet e la forma bilineare $a$ si riferisce alla forma variazionale associata al problema ausiliario agli autovalori, per ulteriori dettagli si veda [Sal04].

**Teorema 1.** *Siano V,H spazi di Hilbert, con H separabile, V denso in H, e tali che l'immersione di V in H sia compatta. Sia a( , ) una forma bilineare in V, continua, simmetrica e debolmente coerciva. Allora*

**(a)** $\sigma(a) = \sigma_p(a) \subset (-\lambda_0, +\infty)$. *Inoltre, se la successione degli autovalori* $\{\lambda_m\}_{m\geq 1}$ *è infinita allora* $\lambda_m \to +\infty$*;*

**(b)** *se u ,v sono autovettori corrispondenti ad autovalori differenti, allora a(u,v) = 0 = (u,v). Inoltre, H ha una base ortonormale* $\{u_m\}_{m\geq 1}$ *di autovettori di a;*

**(c)** *la successione* $\{u_m/\sqrt{\lambda_0 + \lambda_m}\}_{m\geq 1}$ *costituisce una base ortonormale in V, rispetto al prodotto scalare*

$$((u, v)) = a(u, v) + \lambda_0(u, v).$$

$\sigma$ è lo spettro della forma bilineare $a$, $\sigma_p$ il suo spettro puntuale, $\lambda_0$ è l'eventuale costante necessaria per la debole coercività. La base che si ottiene è dunque ortonormale rispetto al prodotto scalare $L^2(\gamma)$.

Osservazione – Caso condizioni al bordo di Dirichlet

Nel caso di condizioni al bordo di Dirichlet il problema si semplifica. Infatti non occorre risolvere l'equazione non-lineare: gli autovalori che si ottengono sono noti a priori e sono della forma

$$\begin{aligned} K_{y,p} &= \left(\frac{\pi p}{L_y}\right)^2 \qquad p = 1, ..., m_y \\ K_{z,q} &= \left(\frac{\pi q}{L_z}\right)^2 \qquad q = 1, ..., m_z \\ \lambda_{p,q} &= K_{y,p} + K_{z,q} \end{aligned} \tag{1.15}$$

Numericamente, o analiticamente come nel caso di condizioni al bordo di Dirichlet, è possibile trovare gli autovalori dei sottoproblemi monodimensionali. La difficoltà stà nel legarli in modo opportuno agli autovalori del problema bidimensionale in modo che venga preservato l'ordinamento crescente degli autovalori 2D rispetto all'indice $k$ della base modale. In particolare è necessario costruire una mappa che leghi ad ogni frequenza $k$ la corretta coppia $(p, q)$, per farlo abbiamo sviluppato un algoritmo che verrà descritto nel capitolo relativo all'implementazione

# Capitolo 2

# Implementazione

L'implementazione è stata sviluppata all'interno dell'ambiente `LifeV`, per la compilazione abbiamo utilizzato `cmake` e, per la gestione del codice, abbiamo utilizzato `git`. Il codice si trova nel branch `20130507_HiMod`. Per la gestione di vettori e matrici e per la risoluzione del sistema lineare sono state utilizzate le strutture disponibili in `LifeV` che sono basate su `Trilinos`, tuttavia non abbiamo approfondito la gestione delle strutture dati e del sistema lineare, che andrebbero sviluppate ad hoc per il metodo in esame: la particolare struttura a blocchi della matrice di sistema non è simile a nessuna delle tipiche matrici degli elementi finiti. Per quanto riguarda la parte di assemblaggio dei sottoproblemi monodimensionali abbiamo invece utilizzato il pacchetto di `LifeV` che sfrutta gli Expression Templates (ETA). Il codice è stato sviluppato in seriale perchè la sua implementazione in parallelo necessita di una struttura dati appropriata.

Prima di procedere con la descrizione dell'implementazione riportiamo le ipotesi di lavoro che sono state fatte per lo sviluppo del codice. Alcune si possono rilassare facilmente, altre invece necessitano un cambiamento più sostanziale.

I - Il dominio di calcolo è un parallelepipedo $(0, L_x) \times (0, L_y) \times (0, L_y)$.

II - Griglia 1D strutturata.

III - Si considera un problema ADR stazionario con condizioni di inflow di tipo Dirichlet e di outflow di tipo Neumann omogeneo.

IV - I coefficienti del problema costanti.

V - Forzante e dato di Dirichlet in ingresso sono generici.

VI - Condizioni sulle pareti omogenee.

VII - È possibile separare il problema lungo le direzioni trasversali, in due sotto problemi agli autovalori.

Come accennato nell'introduzione teorica, la forma del dominio considerato ci consente di utilizzare con semplicità le basi istruite. Nel caso di sezione di forma generica, la gestione di condizioni al bordo sulla parete laterale risulterebbe più complicata. Come sviluppo successivo si potrebbe pensare di utilizzare i polinomi di Legendre al posto delle funzioni trigonometriche. Anche nel caso di generalizzazione dei coefficienti della forma, viene presentata una possibile soluzione, tuttavia il codice è strutturato per l'utilizzo di coefficienti costanti. L'ipotesi sulla griglia 1D strutturata riguarda soltanto alcune funzionalità del post-processing. Per quanto riguarda, invece, l'ipotesi sulle condizioni omogenee a parete, sarebbe necessario costruire un rilevamento dei dati al bordo.

In questo progetto abbiamo considerato un dominio che è un prodotto di intervalli, abbiamo riflettuto sulla possibilità di rendere il codice sufficientemente generale, per poterlo estendere al caso a sezione cilindrica, ma pur essendoci delle somiglianze non ci è sembrato utile costruire una classe astratta da cui ereditare le implementazioni delle due diverse geometrie in quanto era possibile conservare solo l'implementazione di pochissimi metodi.

Per quanto riguarda le condizioni ai bordi di ingresso e uscita il codice permette di applicare condizioni di inflow di tipo Dirichlet non omogenee, le condizione di Neumann all'outflow disponibili sono invece omogenee, ma l'eventuale estensione al caso non omogeneo o ad altri tipi di condizioni al bordo è molto semplice e naturale.

Dall'introduzione teorica si vede come, nella discretizzazione del problema, si fondano due parti molto diverse, da una lato gli Elementi Finiti, lungo la fibra di supporto, dall'altra la base modale 2D, che ricorda molto i metodi spettrali. L'organizzazione delle classi segue questa idea. Per prima cosa è stato necessario costruire delle classi in grado di risolvere i problemi agli autovalori monodimensionali: *Basis1DAbstract* fornisce un interfaccia astratta dalla quale ereditano diverse classi che sono in grado di calcolare gli autovalori e di valutare le funzioni di base. In secondo luogo, la classe *ModalSpace* gestisce il problema agli autovalori bidemensionale, è in grado di ordinare gli autovalori e costruire la mappa tra $k$ e $(p, q)$, fornisce diverse utilità per calcolare coefficienti di Fourier e altro. Essa è la classe che mette in relazione le due basi monodimensionali. Infine il collegamento tra la discretizzazione lungo la fibra di supporto e quella lungo la sezione trasversale è gestito da *HiModAssembler* che rappresenta l'interfaccia esterna che assembla la matrice e il termine noto oltre a contenere alcune utilità ad esempio per il calcolo dell'errore. Altre classi sono state sviluppate per diversi tipi di utilità.

## 2.1 Basis1dAbstract

*Basis1DAbstract* è la classe astratta che definisce l'interfaccia di un generico generatore di basi monodimensionali. Le classi figlie sono degli oggetti pensati per essere utilizzati da *ModalSpace*, essa infatti contiene due puntatori, `M_genbasisY` e `M_genbasisZ`, ognuno dei quali punta a una classe figlia di *Basis1DAbstract*. Le diverse classi figlie di *Basis1DAbstract* si differenziano principalmente per il diverso problema agli autovalori che risolvono. Per ora sono implementate quattro delle nove possibili combinazioni di condizioni al bordo: *EducatedBasis*{*DD,RR,DR,NN*} ognuna di queste risolve un problema con condizioni al bordo di tipo D=Dirichlet, R=Robin e N=Neumann, il carattere di sinistra si riferisce al bordo sinistro dell'intervallo (0,1) il carattere di destra al bordo destro. È anche presente una classe *FakeBasis*, che implementa, con un piccolo trucco, una base finta che, assegnata come base per la direzione $z$ ( o $y$ ) ci consente di utilizzare il software come solutore bidimensionale, è stata sviluppata in fase di debug per testare separatamente i diversi tipi di basi istruite. I problemi agli autovalori sono stati risolti nell'intervallo di riferimento $(0, 1)$ e rimappati nell'intervallo fisico.

### I metodi

Ogni classe figlia eredita pubblicamente da `Basis1DAbstract`

```
class Basis1DAbstract
{public:
    Basis1DAbstract();
    virtual ~Basis1DAbstract();
    void setL (const Real& L);

    virtual void setMu (Real const& mu );
    virtual void setChi(Real const& chi);

    virtual Real chi() const = 0;

    virtual Real Next() = 0;

    virtual void
    EvaluateBasis   (MBMatrix_type& phi,
                     MBMatrix_type& dphi,
                     const MBVector_type& eigenvalues,
                     const QuadratureRule* quadrule) const=0;

    virtual Real
    EvalSinglePoint (const Real& eigen,
                     const Real& yh) const = 0;
protected:
    Real M_L;
};
```

ed implementa in modo proprio i seguenti metodi

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

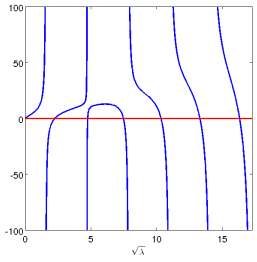
I primi due metodi sono usati da *ModalSpace* in fase di costruzione della base modale, mentre l'ultimo è un'utilità utilizzata nella fase di export gestita da HiModAssembler.

Alcune classi figlie possiedono un membro, `M_ptrfunctor`, che punta a un `EducatedBasisFunctorAbstract`. Abbiamo scelto di costruire un sistema di classi polimorfiche ausiliario strutturalmente simile a `Basis1DAbstract`: le classi figlie si differenziano tra loro in base alle combinazione di condizioni di bordo che devono gestire e, per ognuna di esse, implementano solamente l'operatore parentesi tonde. Sono state implementate *EducatedBasisFunctor*{*RR,DR*}. Il costruttore è di questi funtori 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 ($\mu$, $\chi$ e la lunghezza del dominio fisico 1D). Il funtore rappresenta la funzione analoga a una delle (1.14), ma per il caso in esame. Riportiamo nella seguente tabelle le funzioni utilizzate

| a | b | c | d | Type | $\lambda$ | A | B |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | Dir-Dir | $\lambda_k = (k\pi)^2$ | 1 | 0 |
| 0 | 1 | 0 | 1 | Neu-Neu | $\lambda_k = (k\pi)^2$ | 0 | 1 |
| $\sigma$ | $\mu$ | $\sigma$ | $\mu$ | Rob-Rob | $\tan(\sqrt{\lambda})(\sigma - \frac{\mu^2\lambda}{\sigma}) + 2\mu\sqrt{\lambda} = 0$ | 1 | $\frac{\mu\sqrt{\lambda}}{\sigma}$ |
| 1 | 0 | $\sigma$ | $\mu$ | Dir-Rob | $\tan(\sqrt{\lambda}) + \frac{\mu\sqrt{\lambda}}{\sigma} = 0$ | 1 | $-\tan(\sqrt{\lambda})$ |

Nei casi Dirichlet o Neumann su entrambe i bordi vediamo dalla tabella come, per il calcolo degli autovalori non sia necessario memorizzare alcuna funzione.

Il metodo `Next()` calcola, in successione e uno alla volta, gli autovalori del problema risolvendo un problema di ricerca degli zeri. Esso sarà utiliz-




zato da `EigensProvider()`, un complicato metodo della classe *ModalSpace* che verrà descritto in seguito.

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'obbiettivo 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_dphiy`, 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.

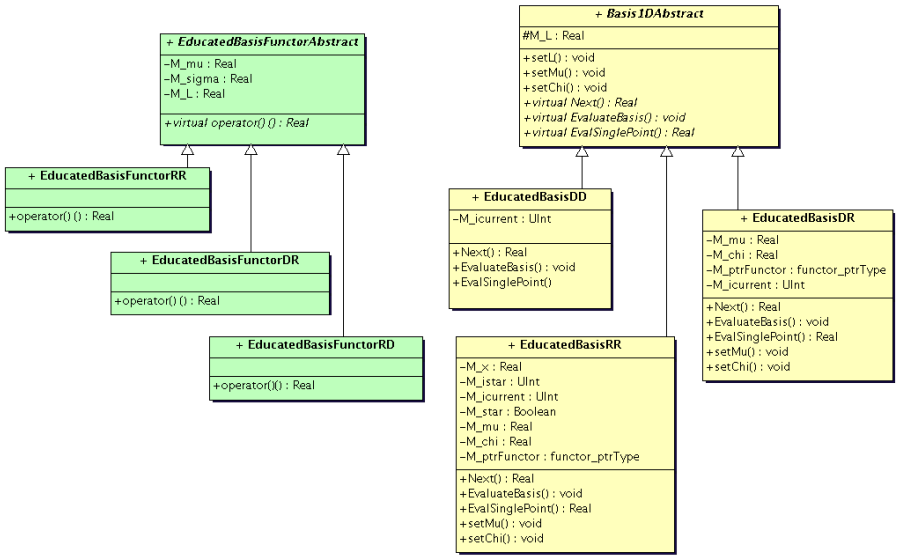



**Figura 2.1:** Classi Basis1DAbstract e EducatedBasisFunctor

## 2.2 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 quantià 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.

### 2.2.1 Costruzione e setting

ModalSpace conosce la geometria della sezione (Ly, Lz) 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 $L_y$.

| Lx | Ly | Max(p,q) | $\| err32 \|_{L_{\inf}}$ | $\| err64 \|_{L_{\inf}}$ |
|---|---|---|---|---|
| 1.0 | 2.0 | 12 | 1.85391e-9 | 9.54883e-14 |
| 1.0 | 4.0 | 16 | 8.7809e-4 | 1.51567e-13 |
| 1.0 | 8.0 | 23 | 7.87756 | 2.70061e-13 |
| 1.0 | 16.0 | 33 | 54.7566 | 7.91844e-6 |
| 1.0 | 32.0 | 50 | 185.541 | 32.7153 |

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::istance().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 [**??**].

$$Aprireildiscorsodellafactory, megliofarloinsieme \tag{2.1}$$

Infine si conlude 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:

$$\begin{aligned} &\hat{\varphi}_j(\hat{y},\hat{z}) = \hat{\eta}_j(\hat{y})\hat{\xi}_j(\hat{z}) \quad \hat{y}\in[0,1] \qquad \hat{z}\in[0,1] \\ &\int_0^1 \hat{\eta}_j^2\, d\hat{y} = 1 \qquad\qquad \int_0^1 \hat{\xi}_j^2\, d\hat{z} = 1 \end{aligned} \tag{2.2}$$

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:

$$\begin{aligned} &\int_0^{L_y}\int_0^{L_z} \varphi_j(y,z)^2\, dydz \\ &= \int_0^{L_y} \eta_j(y)^2\, dy \int_0^{L_z} \xi_j(z)^2\, dz \\ &= \int_0^1 \eta_j(L_y\hat{y})^2 L_y\, d\hat{y} \int_0^1 \xi_j(L_z\hat{z})^2 L_z\, d\hat{z} \\ &= L_yL_z \int_0^1 \hat{\eta}_j(\hat{y})^2\, d\hat{y} \int_0^1 \hat{\xi}_j(\hat{z})^2\, d\hat{z} \qquad \neq 1 \end{aligned} \tag{2.3}$$

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

$$\varphi_j(y,z) = (L_yL_z)^{-\frac{1}{2}}\hat{\eta}_j(yL_y^{-1})\hat{\xi}_j(zL_z^{-1}) \tag{2.4}$$

In conclusione, nei conti che verranno proposti si faccia sempre riferimento all'equazione (2.4).

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 $\hat{z} \in [0,1]$.

### 2.2.2 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 indispendabili ed il loro impiego sarà noto una volta che tratteremo la classe `HiModAssembler`.

### 2.2.3 `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 [**??**]). 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:

$$\lambda_j = (K_y^p)^2 + (K_z^q)^2 \tag{2.5}$$

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:

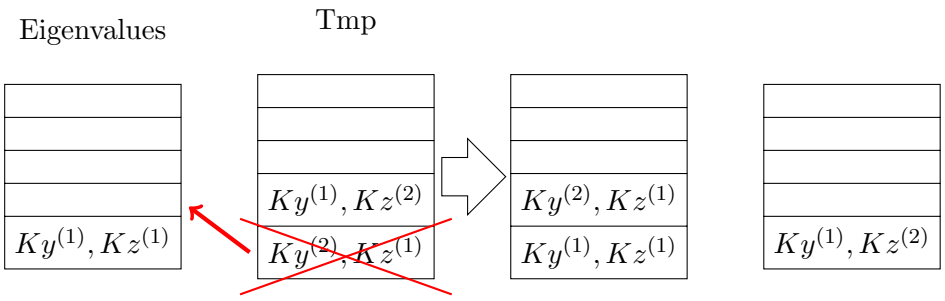
1. Ogni sottoautovalore è il risultato di una ricerca di zeri di una funzione non lineare.

2. L'utente stabilisce il numero massimo di modi sul problema 2D e non sui sotto-problemi 1D.

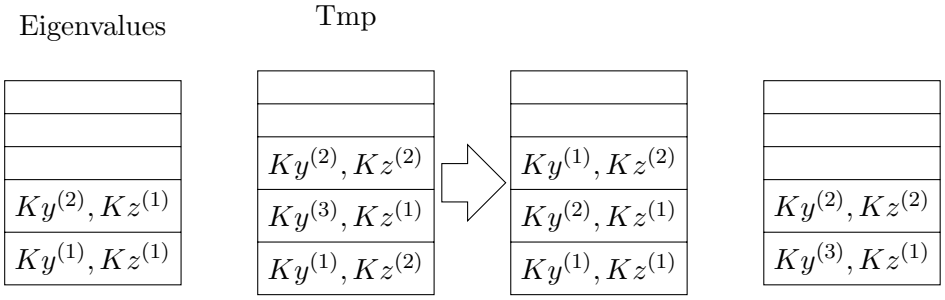
Le due problematiche sono strettamente legate, difatti non siamo interessati a cercare più sottoautovalori del necessario. Si poteva partire calcolando ad esempio 10 sottoautovalori in $y$ e altrettanti in $z$, ordinare la succession 2D e procedere eventualmente nella ricerca. Questo metodo tuttavia 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 utlizzati. 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
```

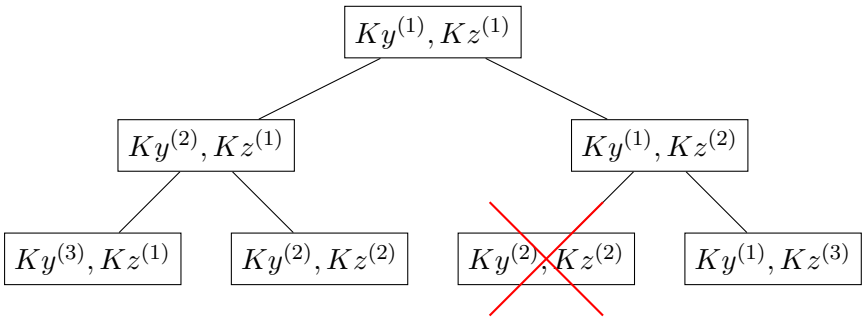



```
-> Cerco Ky_3 e Kz_2 nella mappa o nel registro
     if(trovati) -> li assegno a Ky_3 Kz_2;
     else        Ky_2 = NextY();
                 Kz_2 = Nextz();
-> Inserisco nella mappa la coppia di sottoautovalori minore
     rispetto anche a quelli nel registro
-> Registro i sottoautovalori non inseriti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




```
-> inserimento di copie evitato per unicita' degli oggetti
     nella struttura scelta
```

```
(Ky_2,Kz_2) gi  esistente non viene inserito nella mappa
```




## 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. Momentanemante 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 posside 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 leghi 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 [**??**]), l'estensione a coefficienti non costanti comporta cambiamenti anche nella classe `ModalSpace`, tali modifiche non sono state apportate perchè esuli dai nostri obbiettivi 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 `M_mtot` 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 assemblere il termine noto:

```
boost::shared_ptr<vector_Type> rhs
    (new vector_Typer (Map, Repated));
*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 sitazione piè 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 gerico 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

Nella fase di export risulta necessaria la presenza del metodo `evaluateHiModFunc()`, infatti abbiamo ritenuto opportuno permettere all'utente la valutazione e successivamente l'export, su di una griglia più fitta rispetto al prodotto cartesiano fra nodi di quadratura e nodi di mesh FEM. 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:

$$u(x,y,z) = \sum_{j,s}^{m,n} u_{js}\psi_s(x)\varphi_j(y,z) = \sum_{j,s}^{m,n} u_{js}\psi_s(x)\eta_p(y)\xi_q(z) \tag{2.6}$$

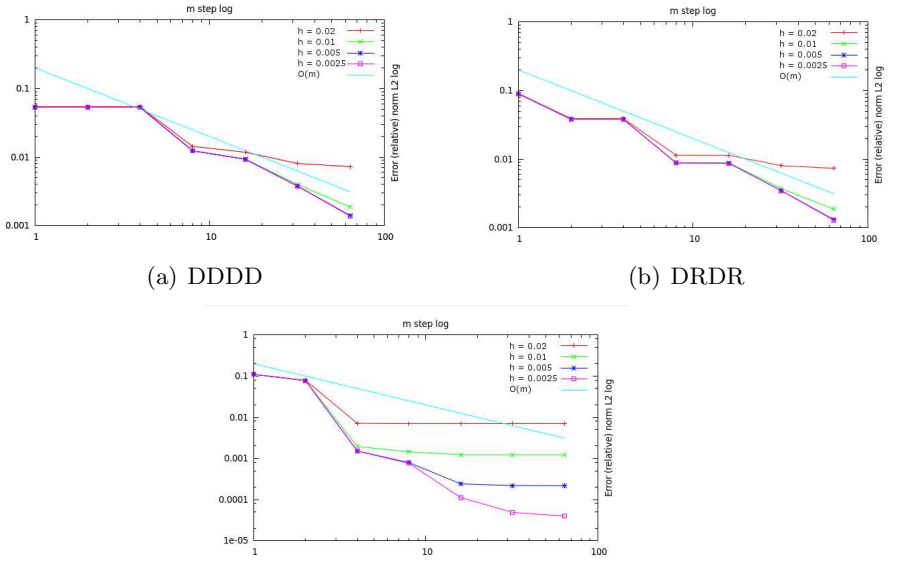
Vi sono due modi di procedere, o fissiamo il modo e calcoliamo tutti i contributi o lo facciamo per la funzione FEM. Noi abbiamo scelto di farlo sui modi, quindi il ciclo più esterno è sui modi. Sarebbe interessante provare a fare l'altro modo per vedere se ci si mette di meno

```
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?)




(a) DDDD

(b) DRDR

(c) RRRR

**Figura 3.1:** Convergenze casi test

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

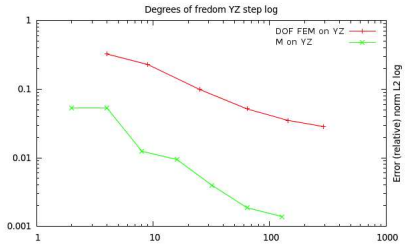



**Figura 3.2:** Confronto DOF

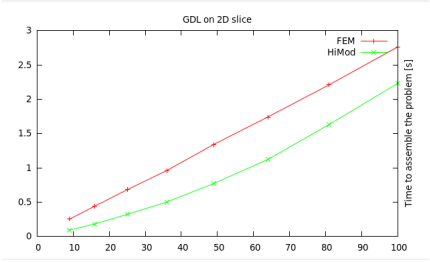
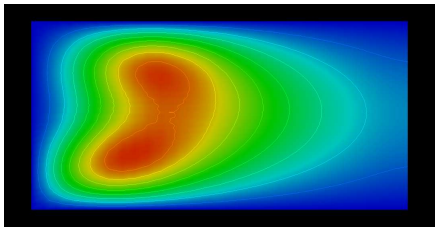
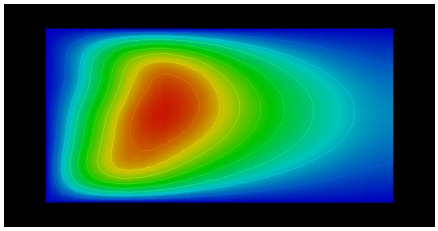



**Figura 3.3:** Confronto tempi assemblaggio

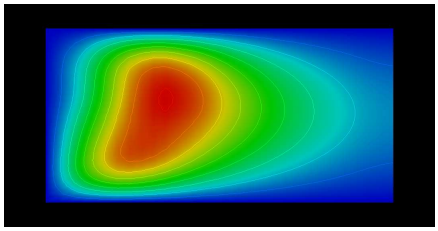
- Presentazione del caso test
- Osservazione sull'aggiunta progressiva di modi (sempre più frequenze vengono considerate e l'approssimazione cresce)
- Commento sui risultati 3D



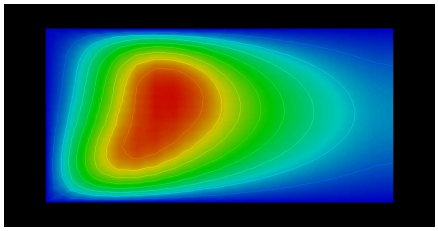
(a) m=9.



(b) m=16.

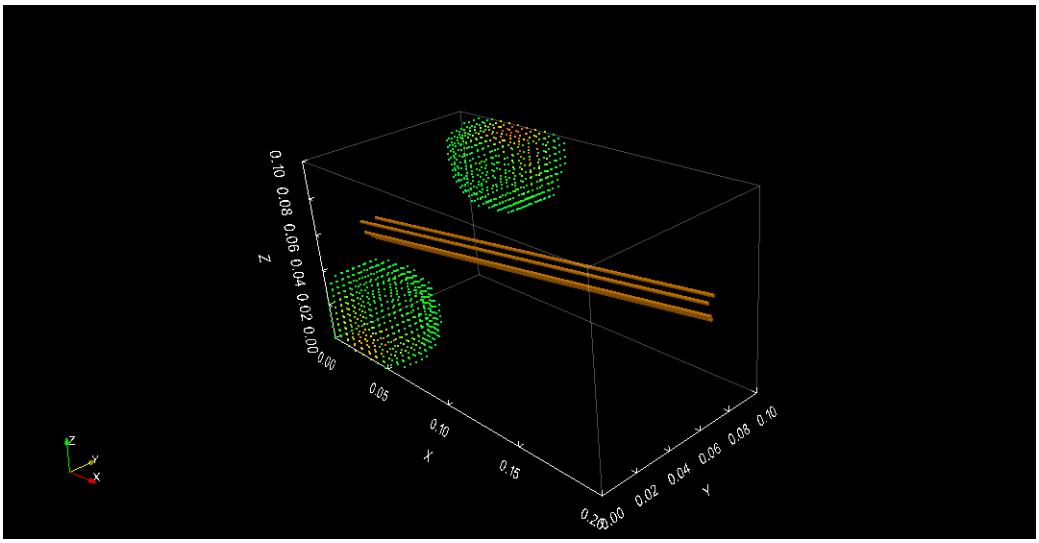


(c) m=25.

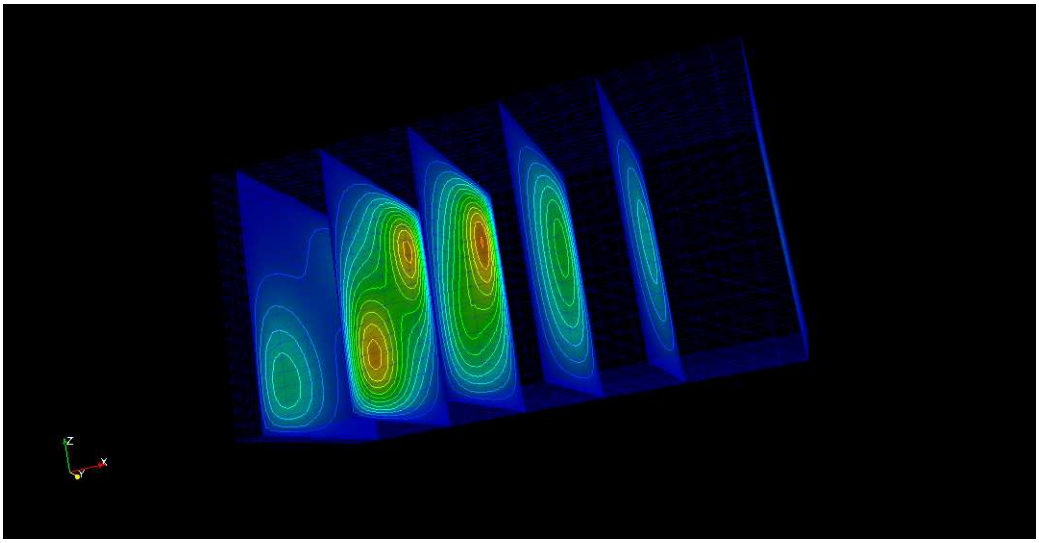


(d) FEM.

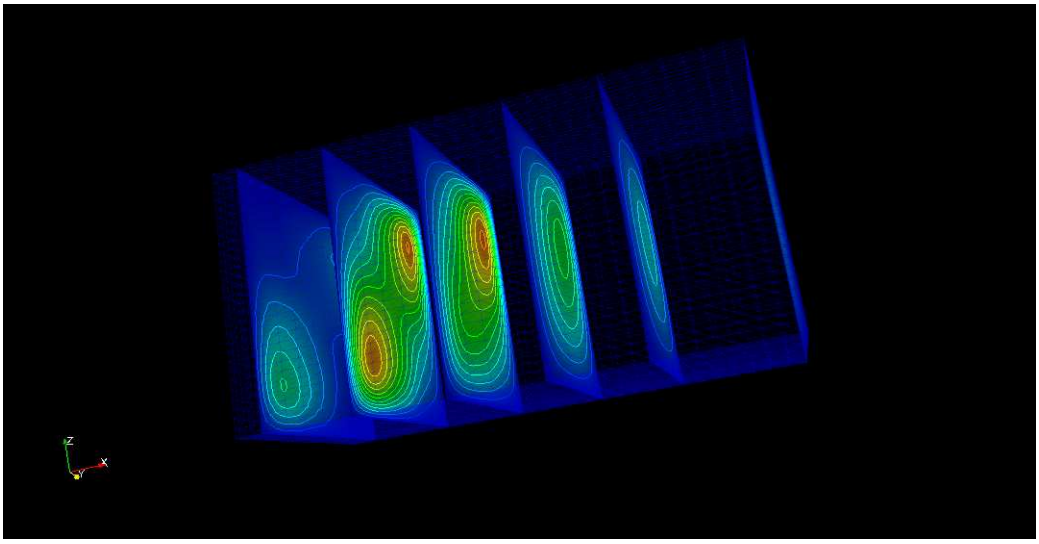
**Figura 3.4:** Caso test DDDD sezione XZ




(a) Forzante e trasporto.



(b) Soluzione FEM.



(c) Soluzione HiMod m=50.

# Bibliografia


[CHQ06] C. Canuto, Y. Hussaini, and A. Quarteroni. *Spectral Methods: Fundamentals in Single Domains*. Scientific Computation. Deutsches MAB-Nationalkomitee beim Bundesministerium für Umwelt, Naturschutz und Reaktorsicherheit, 2006.

[EPV08] Alexandre Ern, Simona Perotto, and Alessandro Veneziani. *Hierarchical model reduction for advection-diffusion-reaction problems*. Mox–report, 2008.

[FQV09] Luca Formaggia, Alfio Quarteroni, and Alessandro Veneziani. Multiscale models of the vascular system. In *Cardiovascular mathematics*, volume 1 of *MS&A. Model. Simul. Appl.*, pages 395–446. Springer Italia, Milan, 2009.

[FSV05] L. Formaggia, F. Saleri, and A. Veneziani. *Applicazioni ed esercizi di modellistica numerica per problemi differenziali*, volume 17 of *Unitext*. Springer-Verlag Italia, Milan, 2005.

[PEV09] Simona Perotto, Alexandre Ern, and Alessandro Veneziani. *Hierarchical local model reduction for elliptic problems I: a domain decomposition approach*. Mox–report, 2009.

[PV09] Joaquim Peiró and Alessandro Veneziani. Reduced models of the cardiovascular system. In *Cardiovascular mathematics*, volume 1 of *MS&A. Model. Simul. Appl.*, pages 347–394. Springer Italia, Milan, 2009.

[PZ12] Simona Perotto and Alessandro Zilio. *Hierarchical model reduction: three different approaches*. Mox–report, 2012.

[Qua08] A. Quarteroni. *Modellistica Numerica Per Problemi Differenziali*. Unitext (En ligne). Springer Milan, 2008.

[Sal04] S. Salsa. *Equazioni a Derivate Parziali: Metodi, Modelli E Applicazioni*. Unitext / La Matematica Per Il 3+2. Springer, 2004.

[VV05] Stefania Vele and Umberto Emanuele Villa. *Modelli 1D per la dinamica del flusso ematico e dei suoi soluti*. Laurea magistrale, A.A. 2004/2005.

[Zil10] Alessandro Zilio. *Modelli anisotropi: analisi ed approssimazione numerica*. Laurea magistrale, A.A. 2009/2010.