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 in emodinamica è possibile incontrare problemi simili dove la direzione del flusso è dominante rispetto alla dinamica in direzione trasversale. L'obbiettivo del 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 convertendo un problema tridimensionale in diversi problemi monodimensionali accoppiati. La riduzione è resa possibile da uno sviluppo in serie di Fourier generalizzate della soluzione solo nella direzione trasversale. L'idea è quindi 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],[PEV10],[PZ13] e in [Zil10] dove il metodo è stato applicato in un contesto bidimensionale (con condizioni di Dirichlet al bordo laterale) e parzialmente sviluppato dal punto di vista teorico nel caso tridimensionale.

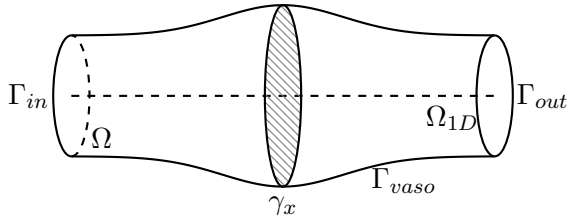
Nel progetto ci siamo focalizzati sull'implementazione del metodo in un contesto tridimensionale con una geometria semplice, ma con diverse tipologie di condizioni al bordo. In particolare abbiamo implementato le basi istruite: una particolare scelta della base di Fourier per la direzione trasversale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

1.1 Descrizione del metodo

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 Ω, visto che il metodo è stato pensato soprattutto per applicazioni in emodinamica, consideriamo un dominio di forma tubolare

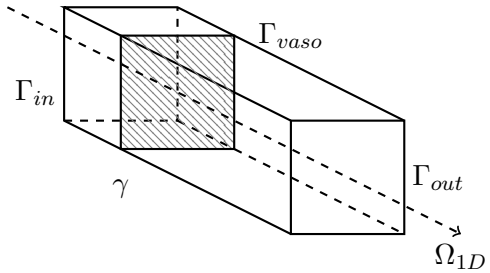
$$\begin{cases} -\mu\Delta u + \mathbf{b}\cdot\nabla u + \sigma u = f & \text{in } \Omega \\ u = u_{in} & \text{su } \Gamma_{in} \\ \frac{\partial u}{\partial \mathbf{n}} = 0 & \text{su } \Gamma_{out} \\ u = 0 & \text{su } \Gamma_{vaso} \end{cases} \tag{1.1}$$



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 Ω in slice poste trasversalmente alla direzione longitudinale

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

Ogni slice viene indicata con γ_x. Lungo γ_x vengono utilizzate funzioni spaziali differenti rispetto a quelle utilizzate lungo Ω_{1D}. Nel caso del tubo a sezione rettangolare, sul quale il codice si focalizzerà, Ω 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 (si veda [EPV08].

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 Ω_{1D} utilizziamo lo spazio $V_{1D} = H^1_{\Gamma_{in}}(\Omega_{1D})$, mentre sulla fibra trasversale γ introduciamo le basi modali $\{\varphi_k\}_{k=1}^{\infty}$ ortonormali rispetto al prodotto scalare di $L^2(\gamma)$. Quest'ultime definiscono su γ 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. Per ulteriori dettagli sulla teoria e sulla notazione si può vedere la tesi di laurea di A. Zilio [Zil10].

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:

$$V_m := \left\{ v_m(x,y,z) = \sum_{k=1}^{m} \varphi_k(y,z)\tilde{v}_k(x), \;\; con \;\; \tilde{v}_k \in V_{1D} \right\}. \tag{1.2}$$

Questo è l'ambiente funzionale in cui opera HiMod. L'ortogonalità in $L^2(\gamma)$ implica che i coefficienti $\tilde{v}_k$ che compaiono nella (1.2) 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 (`20130710_HiModMap`) 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_γ^m *e valga*

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

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:

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

Svolgendo l'operatore gradiente si ottiene:

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

Vediamo come il problema 2D si sia trasformato nella ricerca di m funzioni monodimensionali. Per semplicità consideriamo una partizione τ_h uniforme lungo la fibra di supporto 1D. Sia N il numero di vertici lungo Ω_{1D}. Il passo della partizione è dunque $h = |\Omega_{1D}|/(N-1)$.

Introduciamo lo spazio agli elementi finiti lungo Ω_{1D}

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

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

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

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
Trovare $\mathbf{u} \in \mathbb{R}^{N \cdot m}$ *tale che*

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

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

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

Integrando prima lungo la fibra trasversale si ottiene:

TODO

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

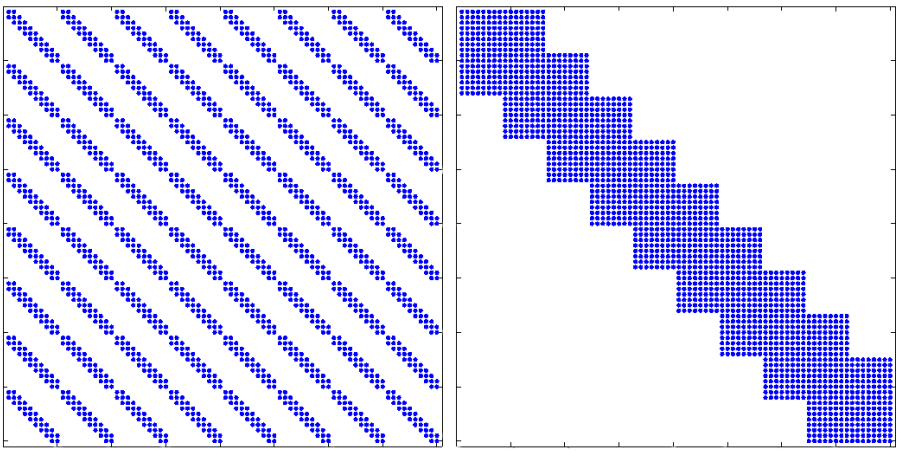


Figura 1.1: Pattern di sparsità per un caso con 9 elementi P1 e 8 modi.
A sinistra il pattern che è stato poi utilizzato, a destra l'alternativa.

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 implementato la prima scelta, che è quella suggerita in [EPV08], 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 γ.[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 sull'intera parete del vaso, dovremo considerare il seguente problema ausiliario

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

Si passi ora al problema agli autovalori associato al precedente sistema e ipotizzando la separazione di variabili $u(y,z) = \varphi(y)\vartheta(z)$, si arriva facilmente ai seguenti sottoproblemi agli autovalori

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

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

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

Esempio - Robin BC

La generica soluzione dei sottoproblemi ottenuti è

$$\begin{aligned} \varphi(y) &= A_y sin(\sqrt{K_y}y) + B_y cos(\sqrt{K_y}y) \\ \vartheta(z) &= A_z sin(\sqrt{K_z}z) + B_z cos(\sqrt{K_z}z). \end{aligned} \tag{1.3}$$

[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 $w = \sqrt{K_z}$)

$$\begin{aligned} f(x) &= 2\mu x + tan(L_y x)\left(\chi - \tfrac{\mu^2 x^2}{\chi}\right) \\ f(w) &= 2\mu w + tan(L_z w)\left(\chi - \tfrac{\mu^2 w^2}{\chi}\right). \end{aligned} \tag{1.4}$$

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*

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

σ è lo spettro della forma bilineare a, σ_p il suo spettro puntuale, λ_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

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

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 (μ, χ e la lunghezza del dominio fisico 1D). Il funtore rappresenta la funzione analoga a una delle (1.4), ma per il caso in esame. Riportiamo nella seguente tabelle le funzioni utilizzate

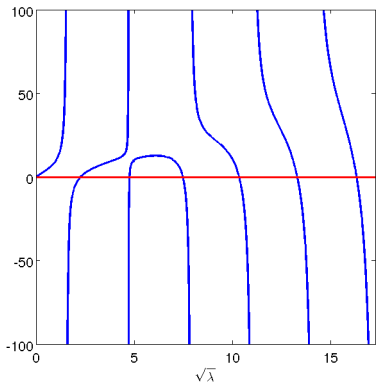
a	b	c	d	Type	λ	A	B
1	0	1	0	Dir-Dir	$\lambda_k = (k\pi)^2$	1	0
0	1	0	1	Neu-Neu	$\lambda_k = (k\pi)^2$	0	1
σ	μ	σ	μ	Rob-Rob	$\tan(\sqrt{\lambda})(\sigma - \frac{\mu^2\lambda}{\sigma}) + 2\mu\sqrt{\lambda} = 0$	1	$\frac{\mu\sqrt{\lambda}}{\sigma}$
1	0	σ	μ	Dir-Rob	$\tan(\sqrt{\lambda}) + \frac{\mu\sqrt{\lambda}}{\sigma}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

Tabella 2.1: Differenti tipi di basi istruite

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

Vedremo che *ModalSpace* possiede delle matrici per contenere la valutazione delle funzioni di base e delle loro derivate nei nodi di quadratura sul quadrato riferimento. Per riempire queste matrici *ModalSpace* chiama i due metodi `EvaluateBasis()` dei generatori di base in direzione y e z: sono quelli infatti gli oggetti che conoscono la forma delle funzioni di base.



Facendo riferimento alla forma generica delle basi modali (1.3) e alla tabella 2.1, 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 necessaria per velocizzare le operazioni di integrazione evitando di valutarle ogni volta nuovamente. Il metodo `EvalSinglePoint()` è stato aggiunto per valutare la funzione di base in un singolo punto. Questa funzione si è resa necessaria in fase di export quanto desideravamo valutare la soluzione su una griglia 3D diversa da quella di quadratura. In figura 2.1

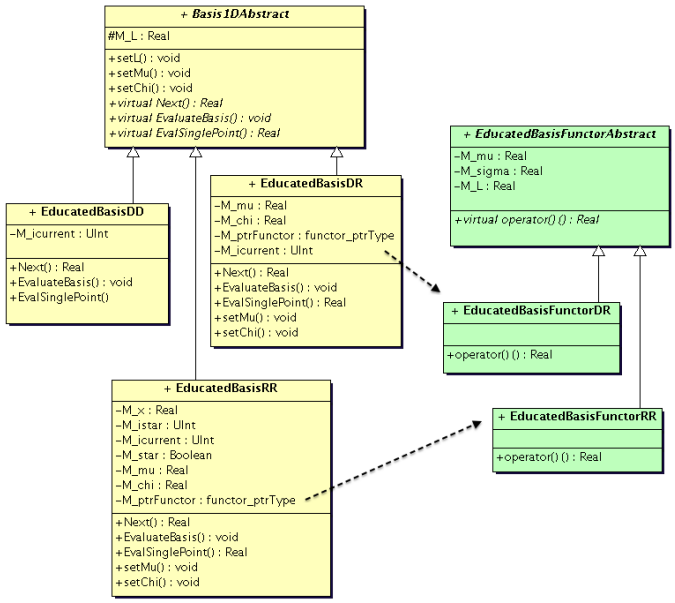


Figura 2.1: Classi Basis1DAbstract e EducatedBasisFunctor

riportiamo una descrizione grafica delle strutture dati appena presentate.

2.2 Modalspace

ModalSpace è la classe che si occupa di utilizzare i due puntatori a *Basis1DAbstract*. Tramite questi oggetti è in grado di calcolare gli autovalori del problema 2D e, successivamente, fornisce diversi metodi per l'integrazione sulla slice e per il calcolo dei coefficienti di Fourier. Inizialmente *ModalSpace* era stata concepita per essere una classe base, dalla quale fosse possibile ereditare ogni possibile scelta delle condizioni al bordo laterale, senza utilizzare *Basis1DAbstract*. Tuttavia facendo un rapido conto ci siamo resi conto che, per implementare tutte le diverse combinazioni di condizioni di Dirichlet, Neumann e Robin sui lati della sezione rettangolare, arriviamo a ottantuno possibili combinazioni, una grande quantità di codice da scrivere, che comprende casistiche molto simili fra loro se non identiche. Abbiamo dunque deciso di scorporare la gestione delle condizioni di bordo dalla classe *ModalSpace*, per poi includerlo in modo ottimale in *Basis1DAbstract*. 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, poteva essere inefficiente. *ModalSpace* contiene le valutazioni delle basi modali e delle derivate su un'opportuna griglia di quadratura, i generatori di basi, diversi metodi di integrazione sulle slice oltre all'algoritmo e alle strutture dati necessarie per la gestione degli autovalori. Questo design, rispetto alla nostra scelta iniziale, attribuisce maggiore generalità: *ModalSpace* è pronta ad utilizzare qualunque tipo di base modale nella forma *Basis1DAbstract*.

2.2.1 Costruzione e setting

ModalSpace conosce la geometria della sezione (Ly, Lz) e deve conoscere il numero di modi da utilizzare, ossia il numero di funzioni di base sulla slice trasversale (mtot). Un altro parametro che si può settare nel costruttore è la regola di quadratura da utilizzare sulla slice. Si noti che le basi utilizzate necessitano regole di quadratura di alto ordine e il numero di nodi di quadratura necessario è strettamente legato al numero di modi[1]. Questo legame è evidente se si pensa che maggiore è il modo, maggiore sarà la frequenza della base modale e il numero delle sue oscillazioni.

Nella sezione quadrata una buona approssimazione del numero di modi in ciascuna direzione è $\sqrt{mtot}$. Il risultato non è valido nel caso di sezioni molto asimmetriche, infatti rettangoli molto allungati in una direzione avranno

[1]Per chi fosse interessato a esplorare questo aspetto c'è un tutorial (`2_check_basis`) che approfondisce questa tematica e consente di fare qualche test

bisogno di più modi lungo la direzione maggiore e meno sull'altra. Per testare qualitativamente la precisione dell'integrazione abbiamo calcolato

$$\sum_{j=1}^{m}\sum_{i=1}^{m}|(\varphi_j\varphi_i)_{numerico}-\delta_{ji}|$$

e abbiamo riportato i valori nella tabella 2.2. Nella seconda colonna della tabella è riportato il massimo tra tutti i valori di p, q. Vediamo come questo valore cresca al crescere dello stretching della sezione: quando il dominio è molto allungato in una direzione tutti i modi vengono spesi per approssimare il fenomeno in quella direzione, mentre nell'altra si userà solo la prima funzione di base (ad esempio q=1 p=1 $\div$ $mtot$). In casi come questo la frequenza massima è altissima ed è dunque meglio utilizzare una regola di quadratura appropriata.

Lx	Ly	Max(p,q)	32 nodi	64 nodi
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

Tabella 2.2: Qualità dell'integrazione al variare della quadratura (mtot=50)

Una volta creato l'oggetto `ModalSpace` bisogna eseguire alcuni set importanti. Per prima cosa dobbiamo settare i generatori di base lungo le direzioni trasversali. Nel caso di basi istruite questa operazione viene fatta imponendo le condizioni al bordo laterale tramite i metodi

- `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(mu);
  M_genbasisY->setChi(chi)
  return;
}
```

Questo metodo si preoccupa di costruire il generatore di base corretto a seconda delle richieste sulle condizioni al bordo. In particolare abbiamo deciso di utilizzare una factory per gestire questo passaggio in maniera dinamica. Tutte le classi figlie di *Basis1DAbstract* devono essere caricate in una factory con un ID ragionevole. Per le basi istruite gli ID hanno la forma $bc_{sinistra} + bc_{destra}$, in questo modo quando l'utente assegna le condizioni al bordo può essere completamente all'oscuro della struttura della basi monodimensionali e della factory: è necessario solamente specificare la natura delle condizioni al bordo. Dopodichè il metodo setta opportunamente i parametri, nel caso di condizioni che non necessitino la definizione di χ o μ si può lasciare il default. Per utilizzare il codice in versione 2D invece che 3D, basta specificare per una delle slice *fake + basis*.

Infine si conlude il setting della classe *ModalSpace* tramite la funzione membro pubblica `EvaluateBasis()`, che chiama le funzioni adibite a riempire le strutture dati che descriveremo successivamente.

Concludendo la parte relativa al setting, riportiamo le uniche linee di codice che l'utente dovrà scrivere nel main.

```
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 una breve descrizione delle strutture dati possedute dalla classe *ModalSpace*. Prima di cominciare vorremmo ricordare che le basi modali sono calcolate sul quadrato di riferimento. Per non incorrere in errori fra dominio reale e riferimento, utilizzeremo la seguente notazione:

$$\hat{\varphi}_j(\hat{y},\hat{z}) = \hat{\eta}_j(\hat{y})\hat{\xi}_j(\hat{z}) \quad \hat{y}\in[0,1] \qquad \hat{z}\in[0,1]$$

$$\int_0^1 \hat{\eta}_j^2\, d\hat{y} = 1 \qquad\qquad \int_0^1 \hat{\xi}_j^2\, d\hat{z} = 1$$

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:

$$\int_0^{L_y}\int_0^{L_z}\varphi_j(y,z)^2\,dydz$$

$$=\int_0^{L_y}\eta_j(y)^2\,dy\int_0^{L_z}\xi_j(z)^2\,dz$$

$$=\int_0^1\eta_j(L_y\hat{y})^2L_y\,d\hat{y}\int_0^1\xi_j(L_z\hat{z})^2L_z\,d\hat{z}$$

$$=L_yL_z\int_0^1\hat{\eta}_j(\hat{y})^2\,d\hat{y}\int_0^1\hat{\xi}_j(\hat{z})^2\,d\hat{z}\qquad\neq 1$$

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

$$\varphi_j(y,z)=(L_yL_z)^{-\frac{1}{2}}\hat{\eta}_j(yL_y^{-1})\hat{\xi}_j(zL_z^{-1}) \tag{2.1}$$

In conclusione, nei conti che verranno proposti si faccia sempre riferimento all'equazione (2.1).

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

- `M_eigenvalues` contiene la mappa fra l'indice della base modale (k) e le sottofrequenze (wp e wq) e i sottoindici corrispondenti (p e q), viene prodotta in fase di setting dello spazio modale tramite la funzione membro `EigensProvider()`, chiamata da `EvaluateBasis()`. Il tipo è un `vector<EigenMap>`.

```
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;
  }
};
```

La componente k-esima del vettore contiene gli indici delle basi modali monodimensionali e le relative frequenze che sono associate agli autovalori dei problemi monodimensionali. L'ordinamento gerarchico degli autovalori e la corrispondenza delle sottofrequenze con i sottoindici sono fondamentali, 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 *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 riempire la struttura dati `M_eigenvalues`, tuttavia il procedimento non è semplice. Per comprendere le difficoltà occorre ragionare sulla struttura del problema: separando le variabili lungo la slice 2D abbiamo ottenuto due problemi agli autovalori 1D (sezione 1.3). Ognuno di questi genera una successione ordinata crescente di autovalori, determinata dalla risoluzione del problema agli autovalori, che nel caso di basi istruite si traduce nella ricerca degli zeri di una data funzione, spesso, non lineare. Definiamo la successione di autovalori in y con $\{K_y\}_p$ e quella in z con $\{K_z\}_q$. Esse definiscono univocamente la successione degli autovalori del problema di partenza 2D e sono in relazione con essa nel seguente modo:

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

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, spesso, 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, combinarli in tutti i modi possibile ottenendo cento autovalori 2D e poi ordinarli. Questo metodo ha 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 sottoautovalori, una volta combinati a formare autovalori 2D, non se ne trovi 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. Il metodo `Next()` dei generatori fornisce progressivamente uno zero alla volta.

L'algoritmo è il seguente

1. Inserire in fondo al vettore degli autovalori 2D (Eigenvalues), al primo posto, la coppia fatta dal più piccolo autovalore di entrambi i sottoproblemi ($p = 1, q = 1$)

2. Calcolare gli autovalori 1D successivi per entrambi i sottoproblemi e inserire gli autovalori 2D associati alle coppie $(p = 2, q = 1)$,$(p = 1, q = 2)$ in un `set` (tmp) ordinato secondo un'opportuna relazione d'ordine che ci consente di evitare doppioni e di avere sempre in cima il più piccolo autovalore 2D.

3. Estrarre da tmp la coppietta $(\bar{p}, \bar{q})$ in testa (associata dunque all'autovalore 2D più piccolo) e inserirla nella lista degli autovalori 2D.

4. Calcolare gli autovalori 1D associati $\bar{p}+1$ e a $\bar{q}+1$, inserire i due nuovi autovalori 2D associati a $(\bar{p}+1, \bar{q})$ e $(\bar{p}, \bar{q}+1)$ in tmp.

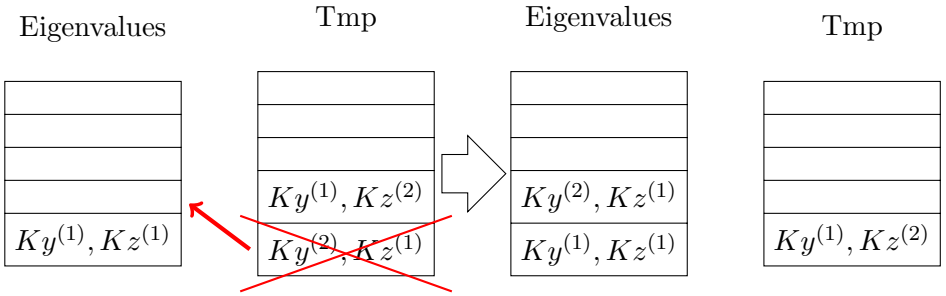
5. Ripetere il punto 3 fino a trovare `M_mtot` autovalori 2D.

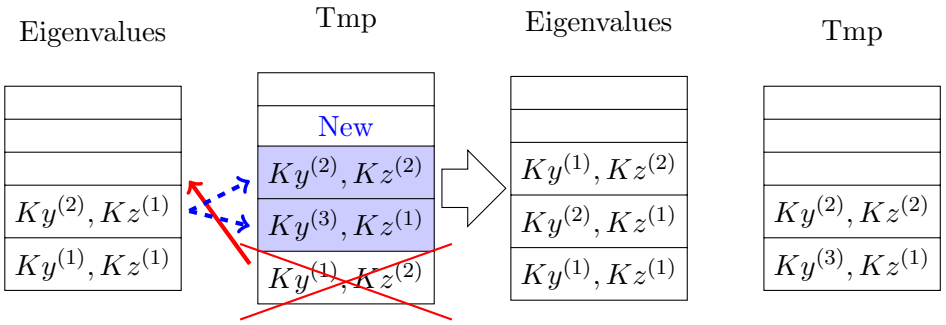
Alcune accortezze sono state prese per evitare di calcolare più volte gli stessi autovalori. La relazione d'ordine è stata scelta con estrema cura per poter considerare autovalori 2D con molteplicità algebrica maggiore di uno, questi autovalori infatti devono essere inseriti in tmp perché non sono doppioni.

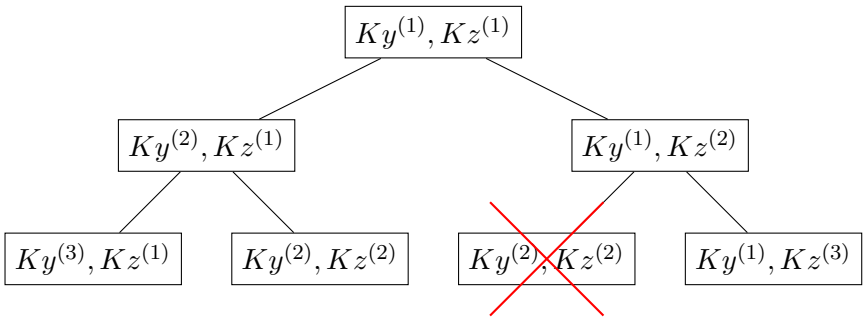
```
bool Comparison::
operator() (const EigenMap& a, const EigenMap& b) const
{
//Se le coppiette non sono associate allo stesso autovalore 2
    D
if ( ( pow (a.wp, 2) + pow (a.wq, 2) ) != ( pow (b.wp, 2) +
    pow (b.wq, 2) ) )
{
  //Restituisci la coppietta associata all'autovalore 2D piu'
     piccolo
```

```
    return (pow (a.wp, 2) + pow (a.wq, 2) ) < (pow (b.wp, 2) +
      pow (b.wq, 2) );
}
else
{
  //Se l'autovalore 2D e' lo stesso restituisci quello con la
     frequenza in direzione y minore, in questo modo, se e' un
     doppione, verra' riconosciuto in quanto ne maggiore ne
     minore
  return a.wp < b.wp ;
}
}
```

Proviamo a dare una visualizzazione dell'algoritmo.







In questo albero rappresentiamo come si effettua la ricerca e vediamo che il nodo con la x rossa è stato cancellato perché è un doppione. Ogni nodo rappresenta un coppietta di sotto autovalori. Ogni volta che un nuovo autovalore viene inserito nella lista di eigenvalues, i suoi due nodi figli

vengono inseriti nell'albero. A ogni iterazione tmp contiene tutte le foglie, ordinate in base al valore dell'autovalore 2D associato, e si occupa anche di non inserire nell'albero i doppioni.

2.2.3 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)`

  $\int_\gamma \varphi_j(y,z)\varphi_k(y,x)\,dydz$

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

  $\int_\gamma \partial_y\varphi_j(y,z)\varphi_k(y,x)\,dydz$

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

  $\int_\gamma \partial_z\varphi_j(y,z)\varphi_k(y,x)\,dydz$

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

  $\int_\gamma \partial_y\varphi_j(y,z)\partial_y\varphi_k(y,x)\,dydz$

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

  $\int_\gamma \partial_z\varphi_j(y,z)\partial_z\varphi_k(y,x)\,dydz$

- `Real Compute_Phi(const UInt& k)`

  $\int_\gamma \varphi_k(y,x)\,dydz$

- `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 FourierCoeffPointWise ( const Real& x,const function_Type& f,const UInt& k ) const`,

  restituisce il k-esimo coefficiente di Fourier di una generica funzione 3D rispetto alla base modale una volta fissato un punto x.

Vediamo come esempio l'implementazione di `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;
  }
```

L'implementazione è basata sulla separazione di variabili, nel caso di coefficienti non costanti bisognerà cambiare questa implementazione. I metodi per calcolare i coefficienti di Fourier sono indispendabili ed il loro impiego sarà più chiaro una volta vista la classe *HiModAssembler*.

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.

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

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 estensione a strutture algebriche più adatte al metodo di riduzione gerarchica.

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 strutture algebriche del tipo EpetraStructured, tuttavia occorrerebbe ripensare

attentamente questa parte, soprattutto in vista di una possibile parallelizzazione. Un codice HiMod ottimizzato dovrà possedere delle proprie strutture, adeguate al pattern della matrice di sistema.

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) si veda [EPV08]. I conti mostrano che alla forma variazionale si aggiungono alcuni casi non gestiti dal `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`. Lo spazio elementi finiti viene comunque conservato per alcune sue utility non presenti in ETFespace di cui avevamo bisogno.

```
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, l'estensione a coefficienti non costanti comporta cambiamenti anche nella classe `ModalSpace`, tali modifiche non sono state apportate perchè esuliano dai nostri obbiettivi progettuali. Si è ragionato su una possibile implementazione che segua la linea generale proposta nel report [EPV08] [2]. 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 ldue frequenze.

```
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 `FourierCoeffPointWise()` posseduto da *ModalSpace*. Calcolare le componenti di una generica funzione H^1 rispetto allo spazio del metodo HiMod richiede prima una proiezione lungo le basi modali e successivamente un'interpolazione dei coefficienti di Fourier nello spazio elementi finiti. Le dimensioni del vettore che contiene le componenti della funzione 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

[2] Nel caso si fosse interessati all'implementazione parte del codice è già presente nel branch `20130710_HiModMap`

`AddADRProblem()`: 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 $\texttt{M\_mtot}^2$ 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) di ogni sottoblocco diagonale. Analogamente l'intervento sul termine noto, viene eseguito sostituendo al primo elemento di ogni sottoblocco, il coefficiente di Fourier del dato in ingresso, moltiplicato per il medesimo coefficiente di penalizzazione usato nella matrice di sistema. Otteniamo dunque che:

$$\tilde{u}_k(0) = \tilde{g}_k \quad \forall k = 1 \dots m_{tot}$$

dove $\tilde{g}_k$ è il k-esimo coefficiente di Fourier del dato all'inflow $g(y, z)$. Per calcolarli è stato sviluppato il metodo `FourierCoefficients()` contenuto in `ModalSpace`.

Analisi

I seguenti metodi si occupano di manipolare le informazioni contenute nel vettore soluzione o in un generico vettore dello 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.

```
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 di base, 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 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.

Export

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

L'export di un generico vettore dello spazio HiMod viene gestito dal metodo `ExporterStructuredVTK()`. Poichè per sua natura la classe HiMod non possiede una griglia 3D definita sul dominio (fatta eccezione la griglia dei nodi di quadratura), viene generata all'interno `ExporterStructuredVTK()` una griglia di supporto su cui eseguire l'export. Utilizziamo il metodo `regulaMesh3D()`. La valutazione delle funzioni test al di fuori dei nodi di quadratura richiede la chiamata a `EvalSinglePoint()`, metodo membro dei generatori di base.

di 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,
                             string content)
```

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

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

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 ExporterFunctionVTK ( const UInt& nx,
                           const UInt& ny,
                           const UInt& nz,
                           const function_Type& fun,
                           const GetPot& dfile,
                           string prefix,
                           string content)
```

Capitolo 3

Risultati

In questa sezione mostreremo alcuni dei risultati che abbiamo ottenuto, per visualizzarli abbiamo utilizzato il software ParaView. All'interno del file `himod/util/CaseTest.hpp` sono salvati diversi casi test con soluzione esatta e uno senza (la forzante è stata ottenuta usando il symbolic toolbox di Matlab). Nel caso si fosse interessati ad aggiungere altri casi test è possibile aggiungerli modificando i file `CaseTest.hpp` e `CaseTest.cpp`. Una volta creato il caso test occorre aggiungerlo allo switch che si trova nella classe *GeneralTest* nella cartella `util`. *GeneralTest* è una classe che abbiamo sviluppato per poter lanciare diversi test senza dover modificare il sorgente, ma semplicimente settando parametri e caso test dal datafile di `GetPot`[1]. Cominciamo mostrando il caso test senza soluzione esatta, che però è più interessante dal punto di vista qualitativo.

$$\begin{cases} -\Delta u + \beta \cdot \nabla u + \sigma u = f & \text{in } \Omega = (0,2) \times (0,1) \times (0,1) \\ u = 0 & \text{su } \Gamma_{in} \cup \Gamma_{lat} \\ \nabla u \cdot \mathbf{n} = 0 & \text{su } \Gamma_{out}, \end{cases}$$

dove $\beta = (5, 1, 0)$, $\sigma = 0.3$ e f è riportata in figura 3.3 e rappresenta due sorgenti di forma sferica. In figura 3.1 abbiamo in alto la soluzione ottenuta con gli elementi finiti (sempre con LifeV), utilizzando una griglia strutturata con 20 elementi in direzione x, y e z, in basso invece c'è la soluzione ottenuta con HiMod, 20 elementi in direzioni x e 50 modi. Osserviamo che 50 modi, in un problema come questo dove la geometria e le condizioni al bordo sono le stesse in direzione x e y, significa circa 7 modi in direzione y e altrettanti in direzione z. Vediamo come, dal punto di vista qualitativo, il fenomeno venga colto bene anche dalla riduzione gerarchica di modello.

Nella figura 3.2 vediamo diverse sezioni longitudinali fissata la coordinata y in tre punti diversi del dominio e possiamo apprezzare, sempre dal punto di vista qualitativo, la convergenza. Vediamo che già con 9 modi

[1] Se si vuole provare a lanciare qualche test si veda il tutorial `4_generaltest`, se invece si vuole vedere come si dovrebbe assemblare un test da zero si veda il tutorial `1_ADR`

la soluzione è ragionevole anche se non è in grado di cogliere bene tutte le caratteristiche della soluzione, con 16 modi ci sono evidenti miglioramenti e con 25 siamo già a convergenza dal punto di vista qualitativo, non abbiamo quindi riportato risultati con più di 25 modi. Infine, in figura 3.4, riportiamo un'altra visualizzazione della sezione centrale con $y = 0.5$. Anche qui possiamo apprezzare la convergenza. Nei successivi esperimenti abbiamo testato diverse combinazioni di dati al bordo e abbiamo cercato di verificare la convergenza del metodo.

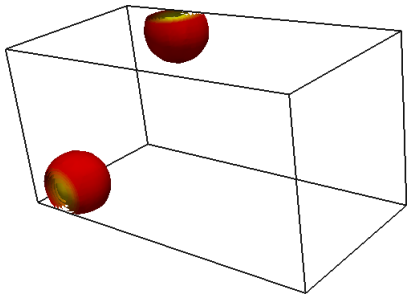
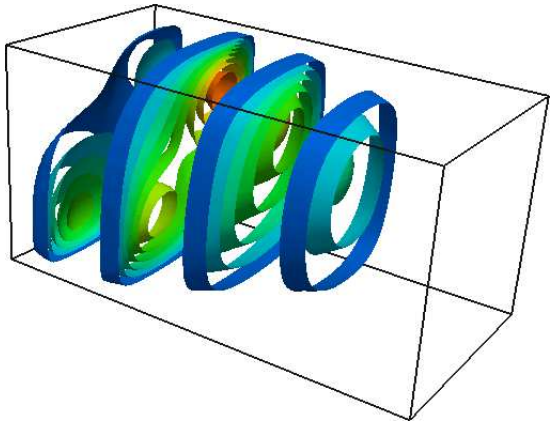
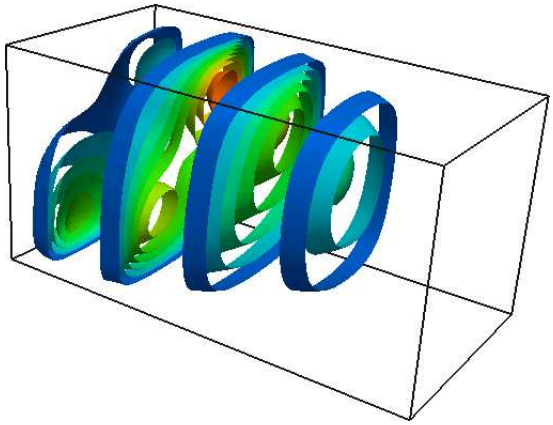


Figura 3.3: Forzante

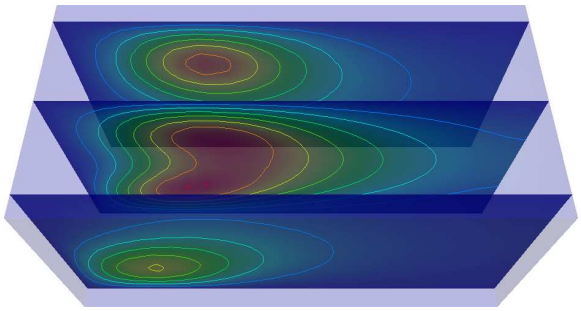


(a) FEM

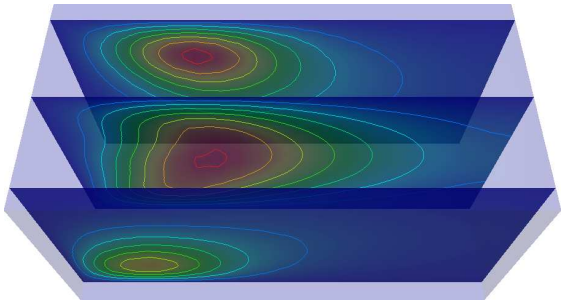


(b) HiMod

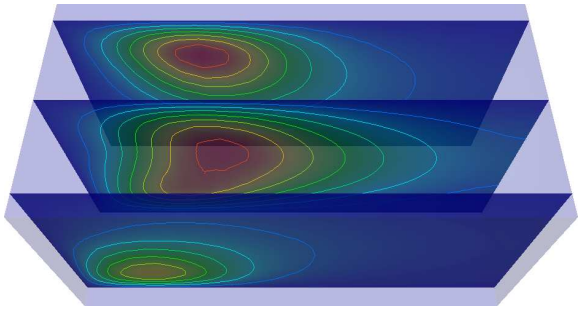
Figura 3.1: Soluzione FEM a confronto con soluzione HiMod, m=50



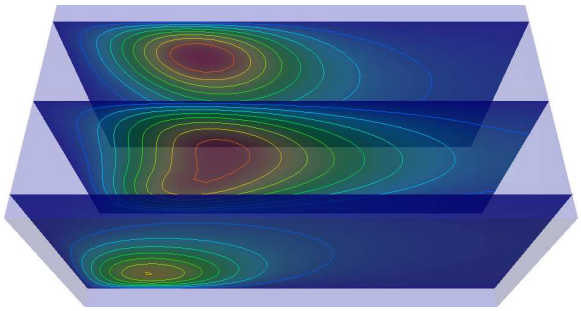
(a) HiMod, m=9



(b) HiMod, m=16

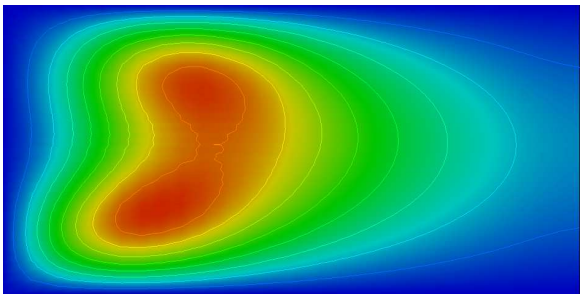


(c) HiMod, m=25

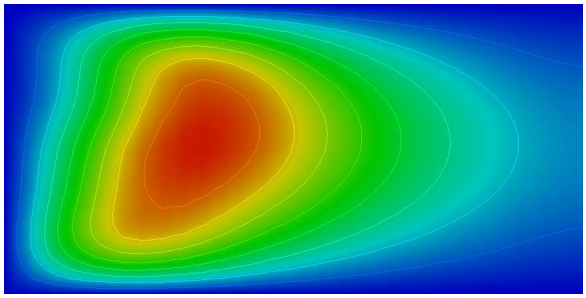


(d) FEM

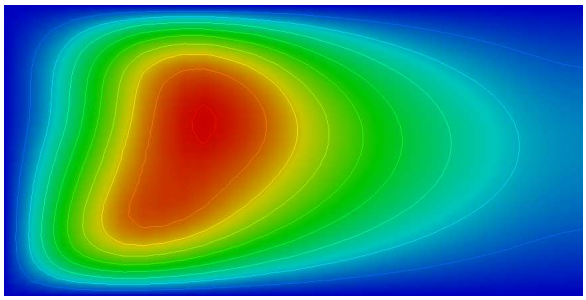
Figura 3.2: Soluzione FEM a confronto con diversi valori di m



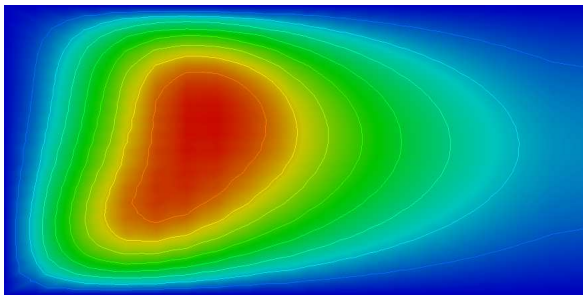
(a) HiMod, m=9



(b) HiMod, m=16



(c) HiMod, m=25



(d) FEM

Figura 3.4: Soluzione FEM a confronto con diversi valori di m

Convergenza

Dal punto di vista teorico la teoria della convergenza per le basi istruite è ancora in fase di sviluppo. Nel caso 2D si ha una convergenza del secondo ordine in $L^2(\Omega)$ rispetto al numero di modi. In particolare nel caso Dirichlet in 2D usando i P1 in direzione x si ha per $u \in H^2(\Omega)$

$$||u - u_{m,h}||_{L^2(\Omega)} \leq C(h^2 + m^{-2})||u||_{H^2},$$

per maggiori dettagli su questo caso si possono trovare in [Zil10] teoremi 3.15 e 3.16.

In 3D tenendo conto che $m \sim m_y \cdot m_x$ e con altre considerazioni basate sulle proprietà del problema agli autovalori è ragionevole aspettarsi un ordine uno in $L^2(\Omega)$ rispetto al numero di modi, ma la dimostrazione non è stata ancora terminata.

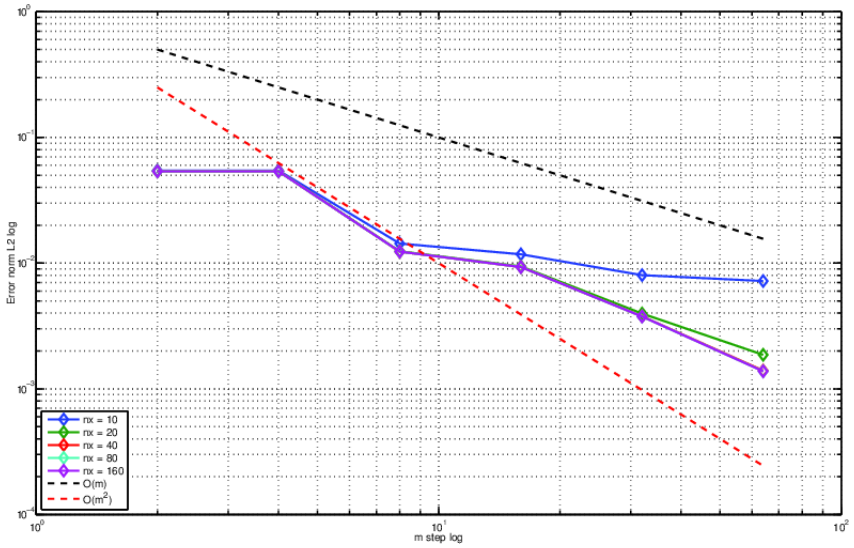


Figura 3.5: Convergenza caso condizioni di Dirichlet

Nella figura 3.5 possiamo vedere un caso test con condizioni di Dirichlet sul bordo laterale. Vediamo come l'ordine di convergenza sia pari a uno. Vediamo anche che se si usa una griglia elementi finiti troppo lasca ad un certo punto l'errore non decresce più al crescere del numero di modi, perchè l'errore elementi finiti è superiore a quello di modello dovuto all'approssimazione modale. Vediamo però che riducendo il passo della griglia le curve dell'errore si attestano tutte sulla stessa linea.

Nel caso riportato in figura 3.6 abbiamo un caso test con condizioni miste sui lati del quadrato: i lati in basso e in alto hanno condizioni di Dirichlet, mentre i lati a destra e a sinistra hanno condizioni di Robin. Anche qui possiamo vedere come il grafico di convergenza confermi i risultati attesi

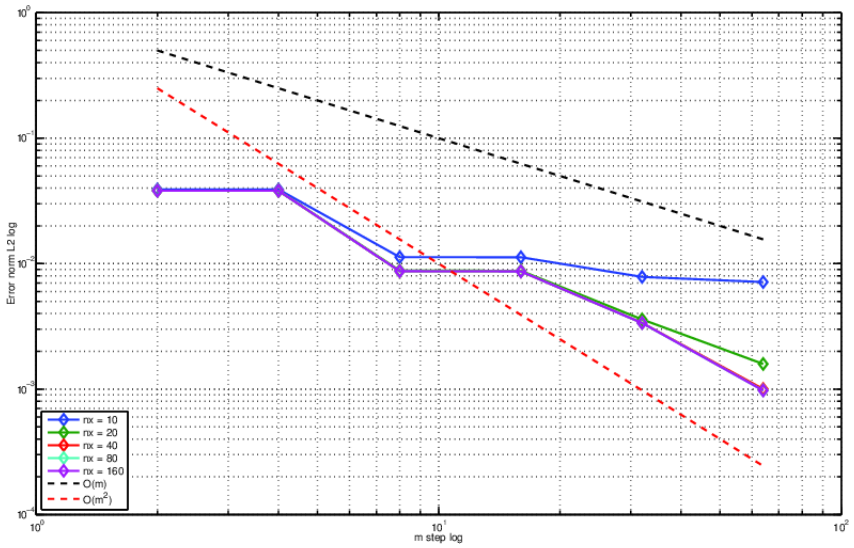


Figura 3.6: Convergenza caso condizioni di Dirichlet e di Robin

dalla teoria. Infine abbiamo costruito un caso test con condizioni di Robin su tutti i lati.

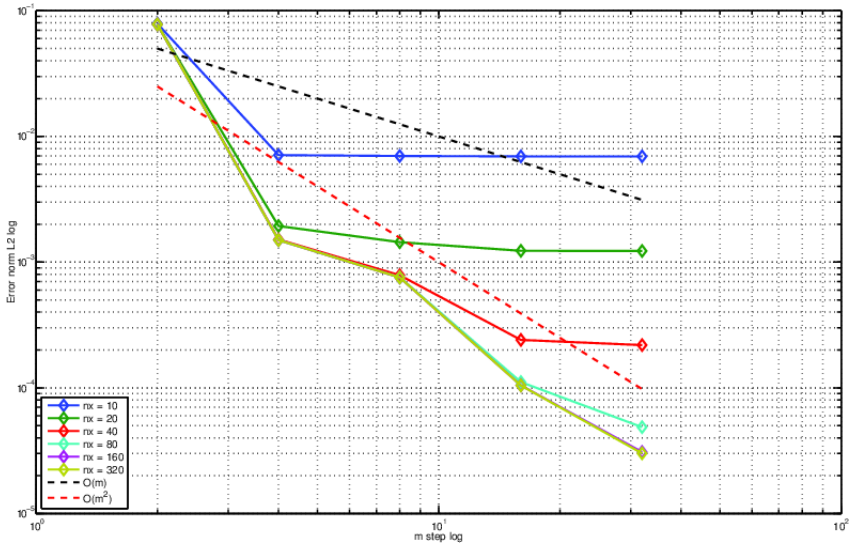


Figura 3.7: Convergenza caso condizioni di Robin

In figura 3.7 possiamo vedere come l'ordine di convergenza sembra essere superiore alla velocità attesa, questo comportamento in realtà si verifica anche nei casi test bidimensionali con condizioni al bordo di Robin, dal punto di vista teorico ancora non ci sono spiegazioni convincenti, tuttavia questo comportamento si verifica puntualmente.

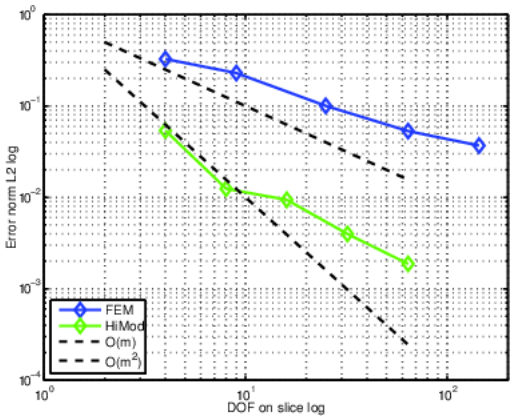


Figura 3.8: Confronti gradi di libertà sulla slice trasversale

Presentiamo inoltre alcuni test per cercare di confrontare gli elementi finiti con HiMod. In figura 3.8 riportiamo sulle ascisse il numero di gradi di libertà utilizzati sulla slice trasversale nel caso test con condizioni di Dirichlet e sulle ordinate l'errore in norma L^2. Per gli elementi finiti abbiamo utilizzato una griglia strutturata ed entrambi i metodi condividono la stessa griglia in direzione x.

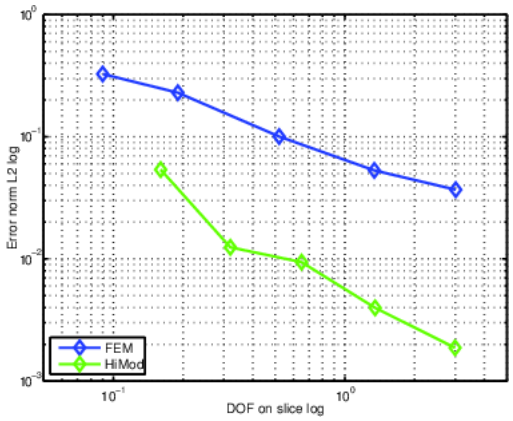


Figura 3.9: Confonto tempi di assemblaggio

Si vede come a parità di precisione, con himod sia possibile utilizzare meno gradi di libertà in direzione trasversale. Tuttavia questo dipende anche dal caso test, perchè l'ordine di convergenza è lo stesso, quindi tutto dipende dalla costante davanti alla stima dell'errore. È chiaro che se la soluzione non presenta dinamiche particolarmente complesse in direzione trasversale, se la base scelta è buona già con i primi modi si potranno cogliere le caratteristiche principali e dunque la curva HiMod in figura 3.8 si troverà al di sotto della curva per gli elementi finiti.

Infine, sempre sul caso test con condizioni di Dirichlet, abbiamo cercato di capire quanto tempo impiega l'assemblaggio della matrice rispetto all'analogo test assemblato con `ADRAssembler` di `LifeV`. I risultati si possono vedere in figura 3.9 e vediamo come, a parità di precisione, il tempo di assemblaggio della matrice sia minore che nel caso elementi finiti, ciò è dovuto al minor numero di gradi di libertà.

Capitolo 4

Conclusione e sviluppi futuri

In questo progetto abbiamo implementato una prima versione del metodo di riduzione gerarchica di modello in 3D. I risultati ottenuti sono in accordo con le aspettative e aiutano lo sviluppo della teoria e, anche se alcune ottimizzazioni sono ancora possibili, i tempi di calcolo sono già buoni. Per la gestione delle condizioni al bordo laterale abbiamo implementato le basi istruite, ma questa non è l'unica scelta possibile e sicuramente è una scelta difficile da generalizzare a geometrie complesse. Durante l'implementazione non ci siamo focalizzati su alcuni dettagli che non presentano difficoltà teoriche o tecniche, che però andrebbero sistematici, quali la generalità sui diversi tipi di elementi finiti e sulla griglia da usare sulla fibra di supporto, oppure tutte le possibili condizioni di bordo in inflow o in outflow.

Abbiamo invece parzialmente esplorato due possibili direzioni di sviluppo, tra loro collegate,

- estensione del metodo a coefficienti non costanti
- estensione del metodo a geometrie differenti.

Il primo caso è gestito nel codice, anche se non ne abbiamo discusso l'implementazione nella relazione, è infatti presente un metodo `AddADRproblem` che prende in ingresso delle funzioni e non dei coefficienti costanti. Tuttavia è un metodo molto lento, sono necessarie alcune semplici ottimizzazioni e bisogna testarne il funzionamento in modo più intenso dato che per ora è stato testato solo con dei coefficienti costanti inseriti all'interno di funzioni. Nel caso si volesse procedere con test più complessi è presente il tutorial numero sei da modificare (`6_nonconst`).

Nel caso dell'estensione a geometrie più complesse bisogna prendere in considerazione la mappa. Essa genererà dei coefficienti non costanti: sarà dunque necessario perfezionare il metodo descritto sopra. L'estensione alla mappa presenterà due problematiche: la prima è che comparirà nella forma bilineare dei problemi 1D un termine dove la derivata spaziale è sulla funzione test e non sulla base associata alla soluzione, tuttavia, avendo scelto di

usare il pacchetto `ETA` per l'assemblaggio, questo non risulta un problema. La seconda problematica riguarda la scelta delle basi sulla sezione trasversale dove si potrebbero utilizzare i polinomi di Legendre.

Altri aspetti invece sono stati considerati solo teoricamente

1. creare una struttura dati adeguata ad ospitare la particolare matrice di sistema

2. implementare la seconda scelta per l'ordinamento della matrice, presentata nell'introduzione teorica, e testare come cambia la velocità del solutore lineare

3. parallelizzare il codice: con la struttura a blocchi si aprono molte possibilità soprattutto nella parallelizzazione dell'assemblaggio, questo è sicuramente il punto più ampio e più promettente

4. implementare il metodo su un tubo a sezione circolare per gestire domini di forma generica con mappe più semplici

5. implementare il metodo con la logica della domain decomposition (si veda [PEV10]).

Bibliografia

[EPV08] A. Ern, S. Perotto, and A. Veneziani. Hierarchical model reduction for advection-diffusion-reaction problems. In Karl Kunisch, Gnther Of, and Olaf Steinbach, editors, *Numerical Mathematics and Advanced Applications*, pages 703–710. Springer Berlin Heidelberg, 2008.

[PEV10] Simona Perotto, Alexandre Ern, and Alessandro Veneziani. Hierarchical local model reduction for elliptic problems: a domain decomposition approach. *Multiscale Model. Simul.*, 8(4):1102–1127, 2010.

[PZ13] S. Perotto and A. Zilio. Hierarchical model reduction: Three different approaches. In Andrea Cangiani, Ruslan L. Davidchack, Emmanuil Georgoulis, Alexander N. Gorban, Jeremy Levesley, and Michael V. Tretyakov, editors, *Numerical Mathematics and Advanced Applications 2011*, pages 851–859. Springer Berlin Heidelberg, 2013.

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

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