Laurea magistrale
in ingegneria matematica

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

Implementazione in LifeV dell'algoritmo di Riduzione Gerarchica di Modello

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

Anno Accademico 2012–2013

Indice

Capitolo 1

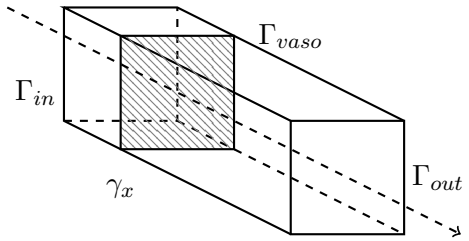
Introduzione

1.1 Nozioni base

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

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

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



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

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

Definiamo ora il sottospazio generato solo dai primi m modi ovvero $V_{\gamma_x}^m := span\{\varphi_1, ..., \varphi_m\}$ 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'ortogonalitá in $L^2(\gamma_x)$ implica che i coefficienti $\tilde{v}_k$ in (1.3) sono il risultato del seguente prodotto scalare per $k = 1, ..., m$:

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

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

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

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

1.2 Forma matriciale

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

$$\int_\Omega \left(\mu \nabla u_m \nabla v_m + \mathbf{b} \nabla u_m v_m + \sigma u_m v_m \right) d\Omega = \int_\Omega f v \, dxdy \tag{1.4}$$

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

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

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

$$\sum_{j=1}^{m}\Bigg[\int_\Omega \mu\nabla(\tilde{u}_j(x)\varphi_j(y,z))\nabla(\vartheta(x)\varphi_k(y,z))\,dxdydz$$
$$+\int_\Omega \mathbf{b}\nabla(\tilde{u}_j(x)\varphi_j(y,z)\vartheta(x)\varphi_k(y,z)\,dxdydz$$
$$+\int_\Omega \sigma\tilde{u}_j(x)\varphi_j(y,z)\vartheta(x)\varphi_k(y,z)\,dxdydz\Bigg]$$
$$=\int_\Omega f\vartheta(x)\varphi_k(y,z)\,dxdydz \tag{1.5}$$

Svolgendo l'operatore gradiente si ottiene:

$$\sum_{j=1}^{m}\Bigg[\int_\Omega \mu(\partial_x\tilde{u}_j\partial_x\vartheta\varphi_j\varphi_k+\tilde{u}_j\vartheta\partial_y\varphi_j\partial_y\varphi_k+\tilde{u}_j\vartheta\partial_z\varphi_j\partial_z\varphi_k)\,dxdydz$$
$$+\int_\Omega (b_1\partial_x\tilde{u}_j\varphi_j+b_2\tilde{u}_j\partial_y\varphi_j+b_3\tilde{u}_j\partial_z\varphi_j)\vartheta\varphi_k\,dxdydz$$
$$+\int_\Omega \sigma\tilde{u}_j\vartheta\varphi_j\varphi_k\,dxdydz\Bigg]$$
$$=\int_\Omega f\vartheta\varphi_k\,dxdydz \tag{1.6}$$

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

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

Possiamo quindi esprimere i coefficienti di Fuorier nel seguente modo:

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

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

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

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

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

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

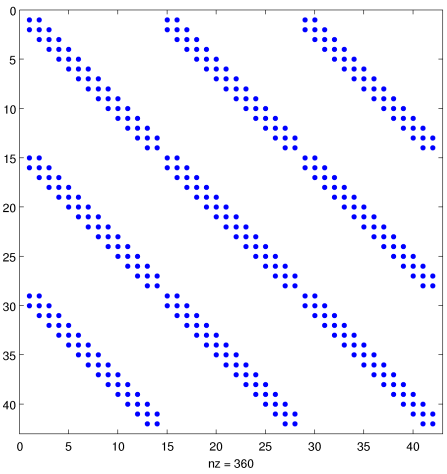


Figura 1.1: Pattern di sparsità per un caso con 14 elementi P1 e 3 modi.

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

1.3 Educated Basis

1.4 Implementazione integrali

Nel caso i coefficienti del problema ADR siano dipendenti dalla sola coordinata x o risultino fattorizabili lungo la direzione x e il piano ortogonale, il risultato finale di HiMod è la trasformazione di un problema ADR full 3D a m^2 problemi ADR 1D accoppiati con coefficienti modificati opportunatamente dalle funzioni modali a seconda della coppia di frequenze considerata. Nel caso non ricadiamo in tale ipotesi vale comunque la scomposizione in problemi 1D ma risulta più delicata l'integrazione. Nel caso si fattorizza anche μ proiettandola sulle basi modali, non penso che dia dei buoni risultati, tuttavia è fattibile

$$\begin{array}{ll} \partial_x\psi_s\partial_x\psi_l & \int_{\gamma_x} \mu\varphi_j\varphi_k \, dydz \\ \partial_x\psi_s\psi_l & \int_{\gamma_x} \varphi_j\varphi_k \, dydz \\ \psi_s\psi_l & \int_{\gamma_x} (\mu\partial_y\varphi_j\partial_y\varphi_k + \mu\partial_z\varphi_j\partial_z\varphi_k + b_2\partial_y\varphi_j\varphi_k + b_3\partial_z\varphi_j\varphi_k + \sigma\varphi_j\varphi_k) \, dydz \end{array} \tag{1.10}$$

1.5 Ipotesi

Le ipotesi alla base di questo progetto sono le seguenti:

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

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

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

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

Capitolo 2

Descrizione classi

2.1 Modalspace

Inizialmente ModalSpace è stata concepita per essere una classe base, dalla quale ereditasse ogni possibile scelta delle condizioni di bordo sulla parete del vaso. Facendo un rapido conto ci si accorge che comprendendo le condizioni di Dirichlet, Neumann e Robin su una sezione rettangolare arriviamo a 81 possibili combinazioni. Una grande 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. Un secondo punto a favore di questa scelta riguarda la valutazione e la lettura delle basi modali. È chiaro che le basi modali sono contenute in ModalSpace, tuttavia ogni figlia avrebbe avuto un tipo di base differente e accedervi tramite la classe base ogni qual volta fosse necessario, non risultava essere efficiente. Infine abbiamo ottenuto maggiore generalità, ModalSpace è pronta ad utilizzare nuovi metodi in grado di generare una corretta base modale, infatti la classe possiede esclusivamente le valutazioni delle basi modali su un'opportuna griglia e i generatori di basi.

2.1.1 Costruzione e setting

ModalSpace conosce la geometria della sezione (Ly, Lz) e sicuramente deve conoscere il grado di precisione 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 di conseguenza si avrà bisogno di una fitta successione di nodi di quadratura. Su una sezione quadrata una buona approssimazione dei nodi necessari su ciascun lato è $\sqrt{mtot}$. Il risultato non è valido nel

caso di sezioni molto asimmetriche, infatti rettangoli molto allungati in una direzione avranno bisogno di più nodi lungo la direzione maggiore e meno sull'altra. Come esempio si osservi la seguente tabella dove sono riportati i check dei valori di normalità di una base, fissata la regola di quadratura al variare della dimensione L_y.

TABELLA (2.1)

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 di future generalizzazioni o aggiunte di nuove basi sarà in questo punto che occorrerà procedere. Qua aprire il discorso della factory?. In pratica si occupano di assegnare il giusto generatore ai membri `M_genbasisY` e `M_genbasisZ`. Nel nostro codice questi membri sono dei puntatori ad oggetti di tipo `Basis1DAbstract`, vedremo nel dettaglio la loro implementazione nella prossima sezione. Infine si conlude il seting della classe ModalSpace tramite la funzione membro pubblica EvaluateBasis(), che chiama le funzioni che si occupano di riempire le strutture dati.

```
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 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 utilizzerà sempre questa forma della base modale.

Procediamo con la descrizione delle strutture dati:

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

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

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

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

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

2.1.2 Metodi di calcolo

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

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

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

  $\int_{\gamma_x} \partial_y \varphi_j(y,z) \partial_y \varphi_k(y,x)\, dydz$

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

  $\int_{\gamma_x} \partial_z \varphi_j(y,z) \partial_z \varphi_k(y,x)\, dydz$

- `Real Compute_Phi(const UInt& k)`

  $\int_{\gamma_x} \varphi_k(y,x)\, dydz$

- `vector<Real> FourierCoefficients (const function_Type& g) const`,

  data una funzione indipendente da x questo metodo restituisce i coefficienti di Fourier (in numero pari ad `M_mtot`) rispetto alla base modale scelta.

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

  restituisce il k-esimo coefficiente di Fourier di una generica funzione 3D valutato nel punto x, rispetto alla base modale.

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

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

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

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

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

  return coeff_y*coeff_z;
```

```
}
```

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

2.1.3 `EigensProvider()`

Abbiamo deciso di dedicare una sezione solamente a questo metodo, poiché la ricerca degli autovalori occupa un ruolo fondamentale nella struttura del codice. Il metodo viene chiamato da `EvaluateBasis()` dunque dopo che sono stati settati i generatori di basi. La funzione deve preoccuparsi di definire la struttura dati `M_eigenvalues`, tuttavia il procedimento non è scontato. Per comprendere le difficoltà occorre ragionare sulla struttura del problema. Separando le variabili della slice 2D si sono ottenuti due problemi agli autovalori 1D. Ognuno di questi genera una successione ordinata crescente di autovalori, determinata dalla ricerca degli zeri di una data funzione. 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:

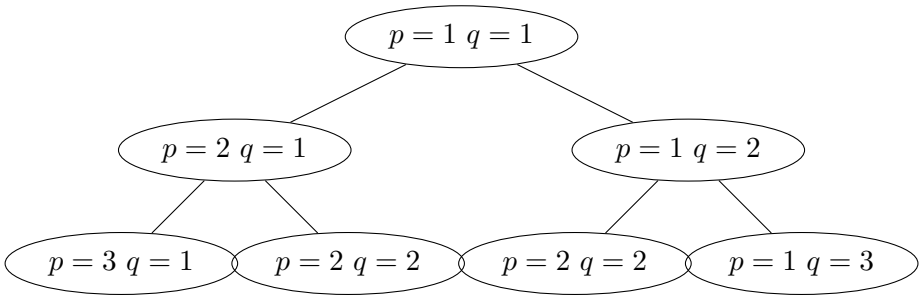
1. Ogni sottoautovalore è il risultato di una ricerca di zeri di una funzione non lineare.

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:

$$ALBERODELLESCELTE \tag{2.6}$$



2.2 Basis1dAbstract

Questa classe si occupa di definire dei generatori di base seguendo la teoria delle Educated Basis (E.B.). Introduciamo con degli esempi l'algoritmo di ricerca delle basi ortonormali, senza entrare nel dettaglio della teoria (per riferimenti si guardi ....):

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

Esempio - RRRR

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

$$\begin{cases} -\Delta u(y,z) = 0 & \text{in } \gamma_x \\ \mu \nabla u(y,z) \cdot \mathbf{n}_{\gamma_x} + \chi u(y,z) = 0 & \text{su } \Gamma_{vaso} \end{cases} \tag{2.7}$$

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:

$$\begin{cases} -\varphi(y)'' = K_y \varphi(y) & \\ \mu\varphi(y)' + \chi\varphi(y) = 0 & \text{per } y = L_y \\ -\mu\varphi(y)' + \chi\varphi(y) = 0 & \text{per } y = 0 \end{cases} \tag{2.8}$$

$$\begin{cases} -\vartheta(z)'' = K_z \vartheta(z) & \\ \mu\vartheta(z)' + \chi\vartheta(z) = 0 & \text{per } z = L_z \\ -\mu\vartheta(z)' + \chi\vartheta(z) = 0 & \text{per } z = 0 \end{cases} \tag{2.9}$$

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

Esempio - RRRR

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

$$\begin{aligned} \varphi(y) &= Asin(\sqrt{K_y}y) + Bcos(\sqrt{K_y}y) \\ \vartheta(z) &= Asin(\sqrt{K_z}z) + Bcos(\sqrt{K_z}z) \end{aligned} \tag{2.10}$$

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

Esempio - RRRR

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

$$\begin{aligned} f(x) &= 2\mu x + tan(L_y x)\left(\chi - \tfrac{\mu^2 x^2}{\chi}\right) \\ f(w) &= 2\mu w + tan(L_z w)\left(\chi - \tfrac{\mu^2 w^2}{\chi}\right) \end{aligned} \tag{2.11}$$

Osservazione

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

$$\begin{aligned} K_y &= \left(\tfrac{\pi p}{L_y}\right)^2 \quad p = 1, ..., m_y \\ K_z &= \left(\tfrac{\pi q}{L_z}\right)^2 \quad q = 1, ..., m_z \end{aligned} \tag{2.12}$$

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

I metodi

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

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

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

Il membro principale di ogni classe figlia è `M_ptrfunctor`. L'oggetto puntato da questo shared_pointer è un `EducatedBasisFunctor`. Abbiamo scelto di costruire un sistema di classe base e derivate ausiliario, simile strutturalmente a `Basis1DAbstract`: la classe base è costituita da `EducatedBasisFunctorAbastract` e vi è una classe figlia per ogni possibile combinazione di condizioni di bordo di un problema 1D, per ognuna di esse è implementato solamente l'operatore `()`. Dunque la funzioni non lineare, la cui ricerca degli zeri fornisce gli autovalori dei sottoproblemi, è contenuta in quesa gerarchia di classi ausiliarie. Si noti che il costruttore è comune a tutte le classi figlie, ed è quindi definito solamente nella classe base. Esso si occupa di settare in maniera corretta i parametri riferiti alle condizioni di bordo applicate (μ e χ).

L'unica eccezione è il caso Dirichlet su entrambe i bordi, nell'osservazione abbiamo sottolineato la trivialità di questo caso, che non necessita la definizione del funtore.

I restanti membri hanno l'utilità di memorizzare quale autovalore è stato calcolato per ultimo, in questo modo il metodo `Next()` è in grado di procedere al calcolo progressivo di tutti gli autovalori desiderati, senza tornare

sui suoi passi. Per comprendere meglio la distribuzione degli autovalori, mostriamo nella seguente figura la forma della funzione non lineare nel caso Robin-Robin.

FIGURA (2.13)

Benchè simili, ogni combinazione di condizioni di bordo deve essere tratta separatamente. Inoltre lo studio sulla ricerca degli zeri ....

Quanto vogliamo entrare nel dettaglio in questo punto??

Abbiamo visto che `ModalSpace` possiede un metodo pubblico `EvaluateBasis()`, in realtà questa funzione chiama separatamente i metodi omonimi, di proprietà dei due generatori di basi. 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 (2.10), 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).

Ricordiamo che la base modale viene valutata solamente nei nodi di quadratura, dunque se siamo interessati a valori delle basi su una griglia più fitta non siamo in grado di calcolarli. Per questo motivo è stato scritto il metodo `EvalSinglePoint()` che dato un sotto autovalore e la coordinata, restituisce la valutazione della base modale in quel dato punto. Questa funzione si è resa necessaria dal momento che in fase di export desideravamo interpolare la funzione su una griglia con più nodi rispetto alla quadratura.

2.3 HiModAssembler

In questa classe viene gestita la fase di assemblaggio del problema e quella di export. Si tratta di una classe template:

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

Il primo paramentro si riferisce alla mesh che verrà passata allo spazio elementi finiti 1D, costruito sulla fibra di supporto. I restanti due paramentri sono riferiti alle strutture utilizzate per la matrice si sistema e il vettore

termine noto. Ricordiamo che la struttura della matrice di sistema è ben nota (vedi figura aggiungere figura), 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. 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.

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 ad una futura estensione del codice. Generalizzando il problema ad una qualunque sezione 2D, occorre introdurre una mappa, i conti fatti (vedi citazione) mostrano che alla forma variazionale si aggiungono numerosi pezzi aggiuntivi e inoltre vi sono casi non gestiti dal pacchetto `GeneralAssembler`.

Dobbiamo dare due dritte sulla mappa?O la lasciamo per ultima come possibile lavoro futuro esprimendo qualche ragionamento fatto?

La dichiarazione nel main è semplice, dato che il costruttore si occupa internamente della costruzione dello spazio `ETA`, è necessario tuttavia passare un comunicatore Non era possibile prenderlo da 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

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. Scorre `systemMatrix` per blocchi e in ognuno di esso computa tramite il metodo `integrate` del pacchetto `ETA` il corrispondente problema 1D. Questa operazione si ripete per ogni coppia di frequenze ovvero `M_mtot`^{2}. Fissata la coppia di frequenza vengono calcolati i valori dei coefficienti della forma tramite i metodi di calcolo della classe `ModalSpace`.

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

Per quanto riguarda l'assemblaggio del termine noto occorre prima interpolare la forzante, ottenuta l'interpolazione l'approccio non è differente da `AdADRProblem()`: il vettore `rhs` possiede `M_mtot` blocchi di dimensione ciascuno pari ai gradi di libertà FEM.

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

Data una forzante, 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.`

Osservazione

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. 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 classica matrice di stiffness di un problema agli elementi finiti 1D, è chiaro che sarà sufficiente intervenire sul primo elemento (nel caso di Dirichlet all'inflow) moltiplicandolo per un numero molto grande (10^{30}) . Adottando questa tecnica siamo costretti ad intervenire anche sul termine noto, infatti approfittiamo di questo metodo per interpolare il dato in ingresso ed assegnare la corretta penalizzazione anche al rhs. Anche nel caso del termine forzante dobbiamo tenere conto delle struttura a blocchi del problema: di fatto è sufficiente intervenire sul primo elemento di ogni blocco del rhs. A questo punto è chiaro il ruolo che ricopre 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

Le funzioni funCoeff_3D non fanno altro che valutare la soluzione o la proiezione di un eventuale funzione lungo le basi modali su una griglia 3D composta dai nodi FEM e quelli di quadratura 2D. Valutare la norma su tale griglia non è sbagliato, ma valutarlo su una griglia più fitta?Ha senso farlo?Non ha senso perchè quanto vado a valutare l'errore in normaL2 devo calcolare un'integrale, dunque non disponendo i pesi di una griglia più fitta non sarei in grado di valutare l'errore.

Dato che abbiamo un metodo che dato un punto 3D e un vettore HiMod valuta il valore di tale funzione in quel punto possiamo utilizzare questo metodo e rendere funcoeff3D più leggibile. Tuttavia evaluateHiModFunc si appoggia ai generatori di base e dunque richiamare tale metodo implica una chiamata più lunga e inoltre la valutazione di una funzione. Invece funcoeff3D si appoggia solamente ai valori calcolati e registrati nelle strutture dati di ModalSpace, chiamata breve e inoltre solo il tempo di un assegnamento.Dato che funcoeff3D deve ciclare su tutti i nodi di quadratura 2D e sui nodi FEM 1D questa differenza di tempo si fa sentire a mio parere....LA FACCIAMO VEDERE?LA VENDIAMO COME SCOPERTA????

```
vector_type evaluateBase3DGrid (const function_Type& fun)

vector_type evaluateBase3DGrid (const vector_type& fun)
```

```
Real normL2 (const vector_type& fun)
```

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

Dato il vettore dei coefficienti di una funzione HiMod la valuta nel punto di coordinate passate.

Export

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

1. Griglia strutturata.
2. Griglia non strutturata.

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

Ricordiamo che:

$$u(x,y,z) = \sum_{j,s}^{m,n} u_{js}\psi_s(x)\varphi_j(y,z) = \sum_{j,s}^{m,n} u_{js}\psi_s(x)\eta_p(y)\xi_q(z) \tag{2.14}$$

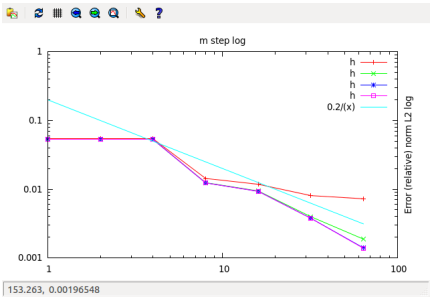
Vi sono due modi di procedere, o fissiamo il modo e calcoliamo tutti i contributi o lo facciamo per la funzione FEM. Noi abbiamo scelto di farlo sui modi, quindi il ciclo più esterno è sui modi. Sarebbe interessante provare a fare l'altro modo per vedere se ci si mette di meno

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

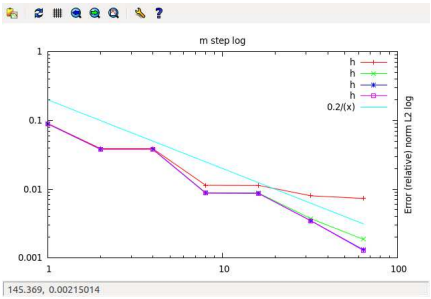
Capitolo 3

Risultati

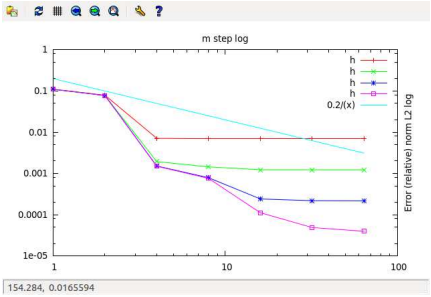
3.1 Convergenza



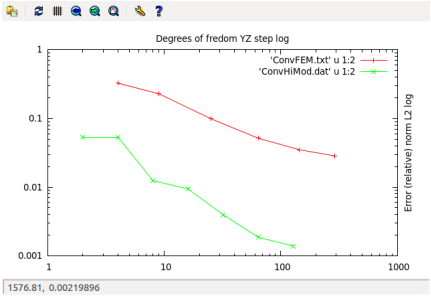
(a) DDDD



(b) DRDR



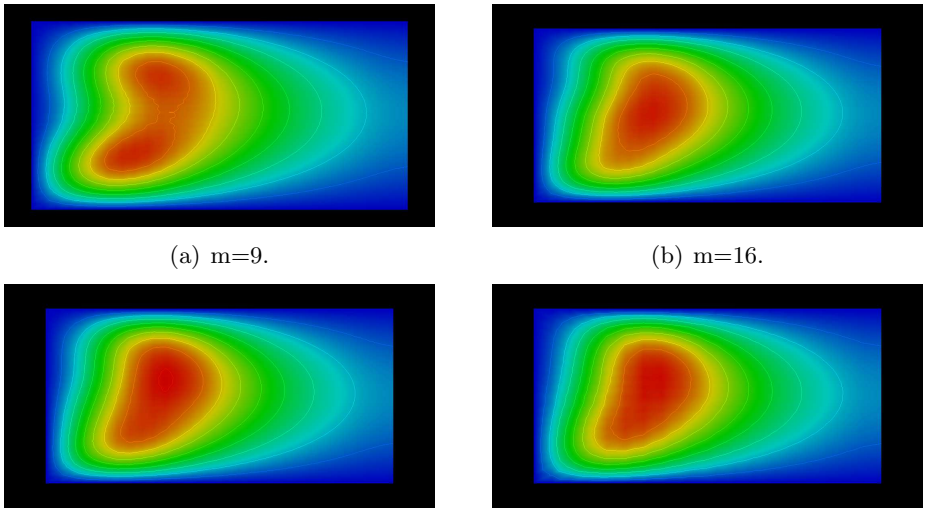
(c) RRRR



(d) dofFEM Vs dofHiMod

Figura 3.1: Convergenze su vari casi test e confronti

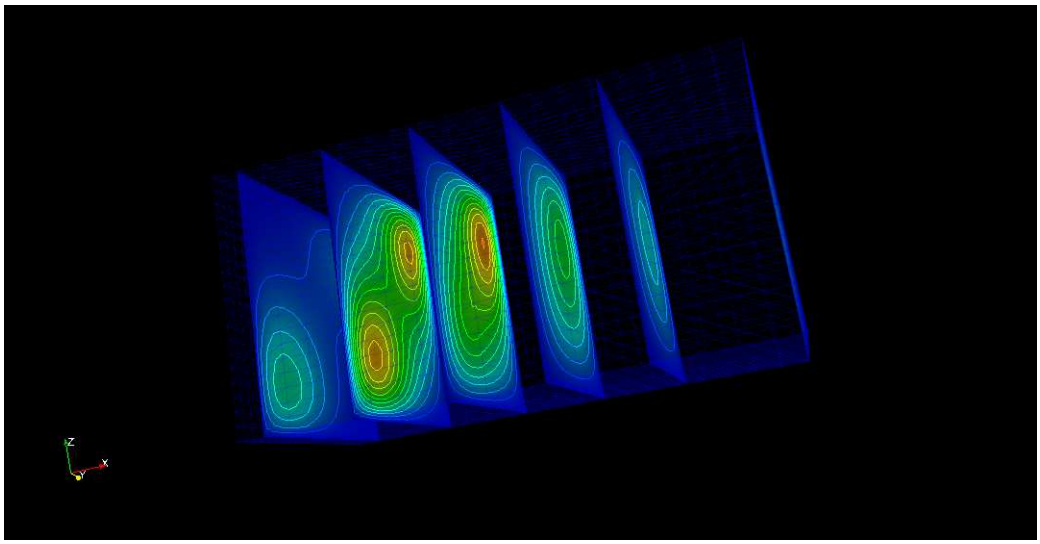
3.2 Foto



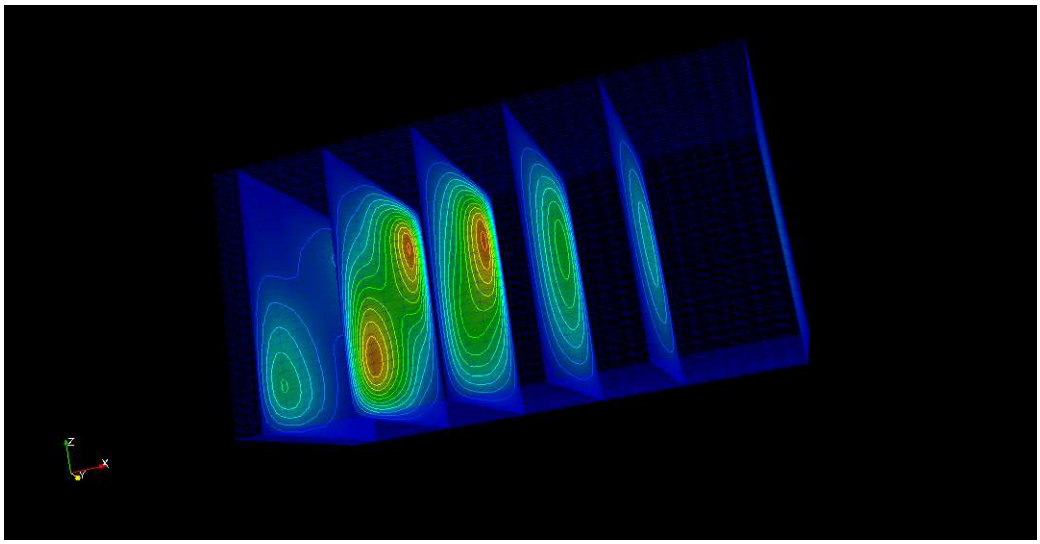
(a) m=9. (b) m=16.

(c) m=25. (d) FEM.

Figura 3.2: Caso test DDDD_ADR sezione XZ



(a) Solusione FEM.



(b) Solusione HiMod m=50.

Figura 3.3: Caso test DDDD_ADR